**TEMA 2 – Arhitecturi OO pentru aplicaţii economice**

# CAPITOLUL 2. Arhitecturi orientate obiect pentru aplicații economice

Conceptele propuse prin abordarea orientată obiect au avut şi au în continuare o mare aderență în privința sistemelor de aplicații economice (business application systems) fiind văzute ca o importantă sursă pentru reprezentarea coerentă, flexibilă şi reutilizabilă a abstracțiunilor extrem de diverse din domeniul economic. Astfel sunt vizate în mod direct reducerea complexității procesului (ciclului) de dezvoltare a aplicațiilor din faza analizei şi proiectării până în faza implementării efective într-un limbaj şi o platformă fundamentate pe principii obiectuale.

## 2.1. Domenii arhitecturale

În această secțiune propunem o „segmentare" a „bagajului" de clase implicate într-o aplicație economică pornind de la o stratificare generică a acestor clase consacrată în domeniul sistemelor construite pe principii orientate obiect. Ulterior, în secțiunile următoare, vom descrie caracteristicile esențiale şi suportul oferit de platforma Java pentru dezvoltarea unor astfel de aplicații.

### 2.1.1. Domenii generice ale claselor de obiecte. Sursele de proveniență

Un sistem de aplicații construit pe o platformă orientată obiect foloseşte sau se sprijină în mod generic pe o „sumă" de clase ce pot fi împărțite sau distribuite în linii mari în cel puțin următoarele domenii: domeniul funcțional specific problemei sau domeniul afacerii (business domain), domeniul arhitecturii de implementare (architecture domain) şi domeniul fundație (foundation domain).

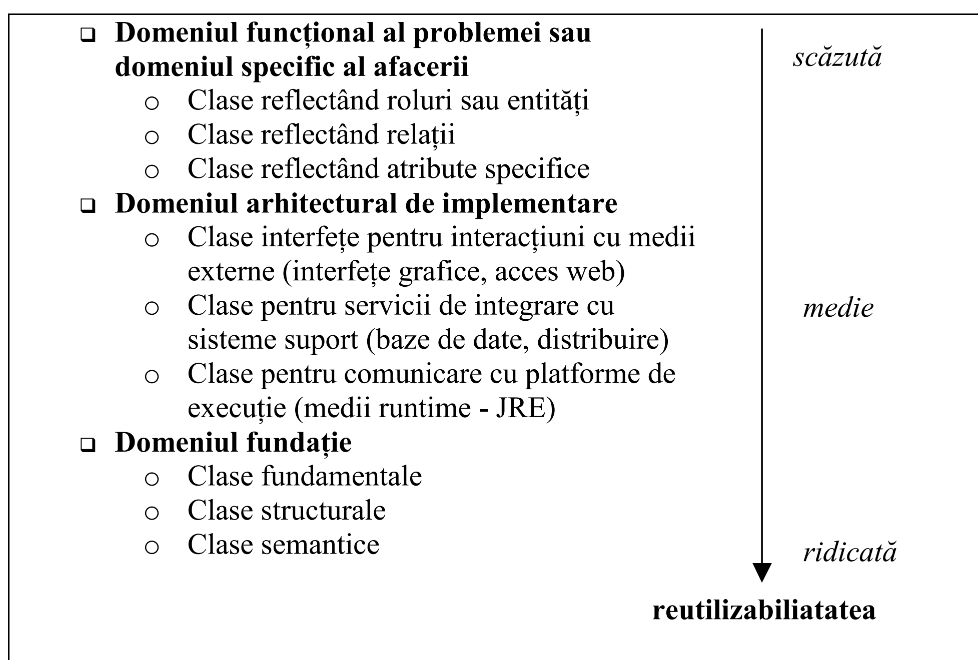



Figure 1 Domenii generice ale claselor de obiecte

Clasele din domeniile specificate mai sus utilizate ca fundament pentru *domeniul specific al aplicației dezvoltate* care formează cadrul de funcționalitate restrâns doar la o singură aplicație. Astfel clasele din celelalte domenii, cu o arie de utilizare evident mai mare decât domeniul specific unei singure aplicații, sunt invocate fie în mod direct (în declarații de variabile şi apel de metode) fie sunt extinse prin clase derivate ce vor particulariza anumite aspecte care țin strict de domeniul restrâns al unei aplicații specifice.

#### 2.1.1.1. *Domeniul funcțional sau specific al afacerii* – business domain

Clasele din domeniul afacerii sunt utile în mai multe aplicații care vizează procese din acelaşi domeniu de activitate (departament, organizație, industrie). Ele sunt împărțite de regulă (funcție şi de metodologia de analiză/proiectare aleasă) în trei categorii:

(a) **clasele entități** care reflectă actorii ce joacă rolurile esențiale din domeniul afacerii, fiind clasele cele mai evidente ca funcționalitate şi resposabilități. Exemple: *Client*, *Comanda, Produs* sau *Student, Disciplina* etc.

(b) clasele ce reflectă **relațiile** sau asociațiile din entitățile afacerii (dacă aceste relații nu sunt reflectate în mod indirect prin atribute de tipul claselor asociate declarate în interiorul acestora), relații sau asociații care pot avea propriile caracteristici şi responsabilități. Exemple: *ProdusComandat* sau *DisciplinaStudiată* (din planul de învățământ).

**(c) Clase-atribute** care nu reprezintă entități sau asociații cu responsabilități proprii, dar care reflectă proprietăți din lumea reală ce au caracteristici ce nu pot fi redate prin tipuri primitive obişnuite. Exemple: *Adresa* (pentru a reflecta o anumită locație de domiciliu sau proveniență) sau *Nota* (pentru exprimarea calificativului obținut la examenul disciplinei studiate).

În mare parte aceste clase vor fi identificare şi structurate în timpul proiectării sistemului informațional aferent domeniului afacerii utilizat fiind relativ independente din punctul de vedere al provenienței față de caracteristicile platformei de implementare.

#### 2.1.1.2. *Domeniul arhitectural de implementare*

Clasele aferente domeniului arhitectural de implementare sunt dependente în general de o arhitectură de calcul particulară fiind reutilizabile în aria aplicațiilor instalabile pe aceeaşi platformă de calcul. Şi în această zonă (dependentă de arhitectura fizică) putem deosebi anumite categorii, dintre care:

(a) *Clase interfețe pentru interacțiuni*, care asigură accesarea (serviciilor) sistemului de către actorii externi direct interesați (utilizatorii şi/sau alte aplicații), de exemplu interfețele grafice sau clasele ce abstractizează suportul de comunicare în rețea (biblioteca Java RMI);

(b) *Clase pentru servicii de integrare cu servicii suport*: asigurarea stocării datelor în medii de persistență cum sunt bazele de date (biblioteca JDBC), asigurarea distribuirii şi integrării (acces şi comunicare) a obiectelor în mai multe locații (rețea) ce asigură

medii de execuție (rezidență – JRE și containere EJB asigurate prin servere de aplicații);

(c) Clase pentru comunicare cu platformele de execuție: abstractizarea resurselor sistemului gazdă: fişiere și directoare, comunicare rețea (porturi, socket-uri)

#### 2.1.1.3. *Domeniul fundație*

Clasele din domeniul *fundație* sunt utile în majoritatea aplicațiilor care folosesc o anume platformă de implementare bazată pe un limbaj de programare specific. Ele sunt utile în mai multe configurații ale unui sistem de calcul (independente de platforma de operare, stratificarea aplicațiilor în rețea, caracteristicile distincte ale domeniului afacerii). Clasele din acest domeniu pot fi încadrate în următoarele categorii:

(a) clase *fundamentale*, care asigurară tipurile fundamentale, predefinite, tradiționale *Integer, Boolean, String*;

(b) clase reflectând *structuri de date*, care implemetează tablourile (array) sau alte tipuri de colecții (*List, Set, Map*);

(c) clase *semantice*, care au o semnificație mai largă decât *Integer* sau *Char* dar care se pot dovedi utile în orice aplicație din orice domeniu al afacerii, de exemplu *Date, Dimension* (reflectând înălțimea și lățimea unei forme), *Point* (reflectând valori într-un sistem de coordonate) sau *Currency, Calendar* etc.

Pentru o exemplificare mai cuprinzătoare putem lua în considerare o aplicație pentru gestiunea angajaților și obținerea statului de salarii. Utilizatorii din departamentul de personal interacționează în mod specific cu aplicația prin intermediul unui client cu interfață grafică tip desktop. Prin urmare va trebui construit unul sau mai multe formulare specifice (*domeniul aplicației*) pe baza suportului oferit de biblioteca conținând kit-ul GUI (*domeniul arhitectural*). Salariații (*domeniul afacerii*) accesibili prin intermediul formularului de personal al aplicației vor fi organizați la nivelul acestuia sub forma unei colecții de tip List (*domeniul fundație*) care permite structurarea sursei de date a unui combo-box (*domeniul arhitectural*) inclus în formular din rațiuni de navigare. Instanțele fiecărui angajat sunt persistente într-o bază de date relațională accesibilă prin intermediul unui driver JDBC (*domeniul arhitectural*). Această succintă descriere poate continua în același mod reflectând faptul funcționalitatea reflectată prin modul specific de interacțiune cu utilizatorii externi se bazează pe un întreg complex de clase furnizate de platforma de implementare (domeniul fundație și arhitectural) și o serie de actori din domeniul afacerii.

### 2.1.2. Modelul "Model-View-Control" specific aplicațiilor Java

În încercare de a reda mai explicit natura stratificării aplicațiilor vom descrie mai întâi un cadru de lucru specific aplicațiilor orientate pe obiecte încă de la începuturile acestora.

#### *2.1.2.1. Caracteristicile generale ale modelului MVC*

Pentru a spori flexibilitatea modelului în privința evoluției sau extinderii viitoare, unul dintre factorii determinanți în modelele dezvoltare propuse pentru aplicații construite folosind platforma Java este cadrul de lucru numit **MVC** (model-view-controler) care îşi găseşte de fapt rădăcinile în lumea *SmallTalk*.

Acest cadru de lucru a fost frecvent utilizat mai întâi în proiectarea aplicațiilor grafice tip *window* bazate pe evenimente, pentru ca apoi să fie adaptat şi pentru cadrul mai larg al structurării sistemelor generice. Componentele implicate în contextul unei aplicații bazate pe obiecte sunt împărțite în:

- componente *model* – obiectele prin care se modelează problema ce urmează a fi rezolvată;
- componente *view* – obiectele care vor determina modul în care vor fi prezentate componentele *model*;
- componente *controller* – obiectele care vor detecta interacțiunile şi vor gestiona fluxul de interacțiuni în care sunt implicate componentele *model*.

Avantajele unei astfel de abordări ar fi următoarele:

- procesarea *intrărilor(input)* este separată de procesarea *ieşirilor(output);*
- *controller-ele* pot fi inter-schimbate asigurând astfel moduri de interacțiune diferite cu actorii externi;
- pot fi implementate mai multe *viziuni (view)* ale modelului.

Relațiile dintre aceste componente ar putea fi prezentate astfel:

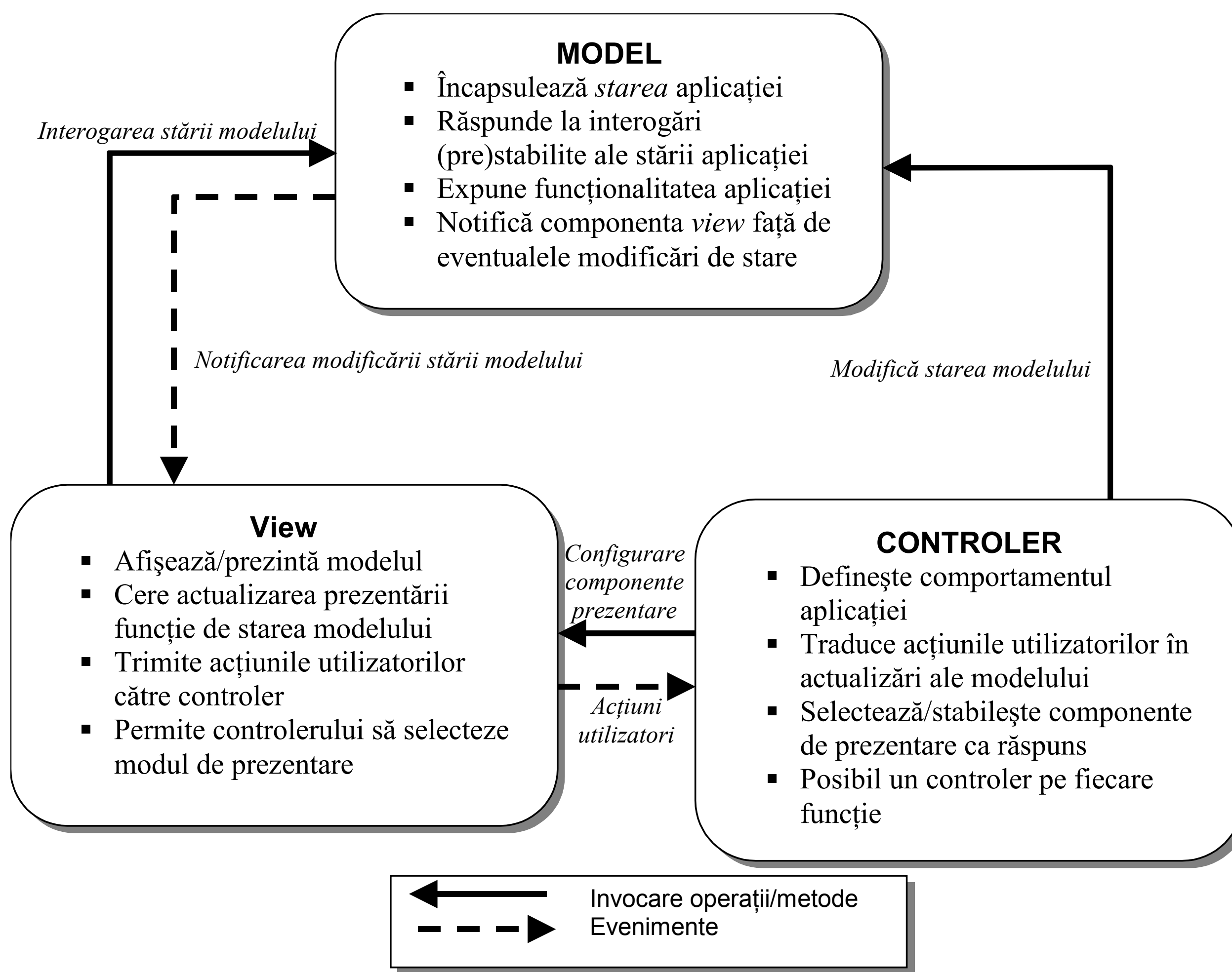



Figura 2-1 Framework-ul *Model-View-Controler*

După cum se poate observa din figura anterioară, componenta *view* şi cea *controller* cunosc în mod explicit *modelul*, însă modelul nu cere nici un fel de detalii față de celelalte componente. Cu alte cuvinte, modelul nu este interesat de modul în care este prezentat sau de modul în care utilizatorii interacționează cu aplicația. Prin urmare, în timp de controller-ul şi view-ul depind în mod esențial de model, modelul nu depinde în nici un fel de acestea.

Responsabilitatea componentei *view* este ca prezentarea modelului să reflecte starea curentă a acestuia. Prin urmare *view-ul* trebuie notificat când modelul îşi modifică starea.

Controlerul trebuie să aplice modificările asupra modelului aşa cum sunt determinate de acțiunile (intrările de date) actorilor externi (ale utilizatorilor). Controller-ul poate, de asemenea, să definească modul în care componenta *view* reacționează la acțiunile utilizatorilor. Componenta view foloseşte controller-ul în primul rând pentru a implementa o strategie de răspuns adecvată acțiunilor utilizatorilor. Ca urmare controlerul captează *evenimentele* generate de componenta *view*.

#### *2.1.2.2. Adaptarea modelului MVC la specificul aplicațiilor economice*

Modelul MVC poate constitui baza „rafinării" claselor implicate în construirea aplicațiilor economice. Caracteristicile definitorii pentru sistemele de aplicații economice rezidă în faptul că accesează şi prelucrează date stocate de regulă în baze de date, iar prelucrările se

desfăşoară conform unui sistem de procese economice ale unei organizații ce impun un set de reguli (business rules) funcție de care se definesc stările de consistență ale datelor.

Structurarea componentelor aplicațiilor economice luând ca reper abordarea MVC se poate face după modelul următor:

- ***model*** – modelul reprezintă datele organizației şi regulile de funcționare ale afacerii (business rules) care guvernează accesul şi prelucrarea (actualizarea) datelor. Cel mai adesea modelul serveşte la o *abstractizare software* a proceselor economice reale, astfel încât tehnicile de modelare ale lumii reale se vor aplica în definirea modelului;
- ***view*** – componentele view for prezenta conținutul modelului: acestea accesează datele organizației prin intermediul modelului şi specifică cum vor fi prezentate acestea utilizatorilor. Componentele view au şi responsabilitatea menținerii consistenței (sau sincronizării) între materializarea prezentării şi modificările care pot surveni la nivelul modelului;
- ***controller*** - controlerul translatează interacțiunile dintre componentele de prezentare (view) în acțiuni ce vor fi efectuate de către model. În cazul unei interfețe grafice (GUI) de sine stătătoare (pentru medii client/server obişnuite) interacțiunile utilizatorilor se pot materializa în selectarea unor opțiuni, modificarea unor valori prezentate printr-o căsuță cu text, apăsarea unui buton etc.

Structura MVC poate fi transpusă la scara aplicațiilor economice astfel: componentele model sub forma *claselor entități*, componentele view sub forma *claselor de graniță*, componentele *controller* sub forma claselor *active*:

- *Clasele entități (entity classes)* reprezintă "baza" conceptelor din domeniul aplicației (de exemplu *Client, Comandă, Produse*). Scopul lor este în primul rând de a reține informațiile despre entitățile persistente şi de a captura serviciile necesare pentru deservirea majorității interacțiunilor din aplicație privitoare la entitatea modelată.
- *Clasele de graniță (boundary classes)* servesc ca element de contact între actorii externi, care doresc să interacționeze cu aplicația, şi clasele entități. Majoritatea claselor de acest tip sunt componente ținând de interfața grafică destinată utilizatorilor şi care se regăsesc ca *formulare* sau *ecrane*. Spre exemplu *formularul de introducere a clienților* sau *formularul de specificare a comenzilor* constituie două componente de acest gen. Ele pot fi implementate ca *JFrame* în cazul unor aplicații Java client/server cu interfață "bogată", sau ca şi compontente *JSP* ce vor genera pagini HTML pentru aplicații Web.
- *Clasele de control* sunt menite să coordoneze activitatea din domeniul aplicației. În unele cazuri *caracteristicile de comportament* nu sunt implementate în nici una din celelalte tipuri de clase, urmând ca toate responsabilitățile în această privință să fie preluate de clasele de control. În mod tipic responsabilitatea claselor de control este legată de:
    - coordonarea între clasele de graniță şi clasele entități (preluarea datelor din clasele entități şi afişarea lor în clasele de graniță, preluarea intrărilor utilizatorilor din clasele de graniță şi modificare claselor entități);

- comportamentul legat de controlul tranzacțiilor;
- servicii care separă clasele de graniță de clasele entități (de exemplu controlul din punctul de vedere al securității fluxului de interacțiuni dintre clasele de graniță şi cele entități).

### 2.1.3. Stratificarea aplicațiilor economice

Pe baza categoriilor de clase definite mai sus putem spune că *stratificare sistemelor de aplicații economice* are în vedere trei categorii de servicii diferențiate prin gradul de incidență al modificărilor (dinamică) şi responsabilități.

Astfel, *serviciile de **prezentare*** au ca scop "livrarea" într-o anumită formă grafică a datelor către utilizatori şi/sau (sub)sisteme externe plus verificări preliminare ale datelor înainte de a fi acceptate în sistem. Tradițional, aceste servicii sunt asigurate prin interfețe grafice utilizator (GUI) sau rapoarte formatate. Serviciile de prezentare sunt „împachetate" sub forma aplicațiilor desktop (clase ***controller***) care prin intermediul unui sistem de formulare şi meniuri pot delimita principalele fluxuri tranzacționale din sistem, şi vor inițializa interfațele grafice (clasele de graniță) interogând entitățile afacerii (prin intermediul serviciilor de implementare a logicii afacerii) pentru a asigura sursele de date expuse prin intermediul controalelor grafice ale formularelor. De asemenea, aceleaşi clase vor gestiona evenimentele produse ca urmare a interacțiunii dintre utilizatori şi clasele de graniță şi vor „depune" efortul coordonării tranzacționale a modificării stării entităților afacerii ca urmare a acestor evenimente.

*Serviciile de implementare a logicii afacerii* (business services) au ca scop aplicarea principalelor reguli funcționale ale sistemului şi aplicarea restricțiilor esențiale privind integritatea datelor. Prin urmare *regulile afacerii* (logica funcțională desprinsă din comportamentul dorit al sistemului modelat) sunt implementate cu preponderență în acest strat de mijloc, completat de *verificările preliminare* din stratul de prezentare, dar şi de restricțiile de la nivelul bazelor de date. Strâns legat de logica afacerii sunt *serviciile privind persistența datelor* care au ca scop în primul rând asigurarea spațiului de stocare durabil şi sigur pentru păstrarea datelor reflectând starea entităților din domeniul afacerii în condiții de integritate maximă, şi reconstituirea acestora. Aceste servicii vor fi implementate cu sprijinul direct (mijlocit prin mijloace de comunicare specifice cum ar fi biblioteca JDBC specifică aplicațiilor Java) al SGBD-urilor.

Prin urmare, componentele Java, rezultate ca urmare a implementării unei astfel de arhitecturi, vor fi localizate astfel:

- formularele sub forma componentelor Swing (JFrame, JDialog etc.) care dau posibilitatea modificării proprietăților *entităților* afacerii sau care generează fluxuri funcționale în sistem, se vor regăsi în stratul de prezentare;
- clasele implicate de modelarea problemei inițiale care implementează entitățile afacerii şi controlează fluxurile funcționale dintre aceste entități, se vor regăsi în *stratul afacerii*;
- clasele implicate în accesul structurilor de stocare din bazele de date pentru reconstituirea obiectelor entități (instanțele claselor entități) precum şi

structurile de stocare specifice SGBD-urilor sunt plasate în *stratul privind serviciile de persistență a datelor*.

## 2.2. Modelarea domeniului afacerii. Notații UML

În continuare vom prezenta modul în care putem formaliza sub forma limbajului de modelare UML entitățile şi relațiile dintre entitățile care formează domeniul afacerii. Limbajul UML nu este limitat la domeniul strict al afacerii, el poate fi folosit pentru a descrie modul de „asamblare" al actorilor din toate celelalte straturi ale sistemului. Cum în ceea ce priveşte dezvoltarea unui aplicații economice de obicei în zona modelării afacerii există cele mai puține „puncte de sprijin" în platforma de implementare, ne vom ocupa în primul rând de acest domeniu.

### 2.2.2. Modelare structurală şi comportamentală în UML

Ca definiție UML reprezintă un limbaj de modelare pentru *specificare, vizualizare, construire şi documentare* a sistemelor rezultate dintr-un proces *software-intensiv*. Prin urmare acest limbaj este gândit pentru a acoperi în întregime cel puțin aspectele de proiectare ale unui sistem de aplicații. În acest sens, conceptele de modelare introduse de UML acoperă următoarele aspecte:

- **Modelarea structurală**: modelarea aspectelor stabile, statice privind sistemul analizat;
- **Modelare comportamentală**: aspectele privind modul în care obiectele interacționează între ele şi modul în care acestea evoluează în timp sau cum îşi pot modifica starea curentă
- **Modelare arhitecturală**: stabilirea modului de stratificare a componentelor.

În continuare vom discuta detaliile ce privesc modelarea structurală ce pot fi traduse în mod transparent într-un limbaj de programe specific unei platforme de implementare, precum şi câteva detalii privind formalizarea grafică a interacțiunilor dintre entități.

#### *2.2.2.1. Modelare structurală*

Modelarea structurală se referă, aşa cum am menționat şi mai înainte, la aspectele stabile, statice al unui model privind sistemul (sau problema) analizată. Prin urmare conceptele şi formalismele fundamentale se referă la clase/obiecte şi relațiile/legăturile dintre acestea.

**Clase, atribute, operații**

*O* **clasă** *reprezintă descrierea unui set de obiecte care partajează aceleaşi atribute, operații, relații şi semantică.* O **clasă** poate fi înțeleasă ca un tipar [sau şablon – stencil] prin care sunt create obiectele (instanțele). Fiecare obiect are aceeaşi structură şi comportament ca şi cel al clasei din care este instanțiat.

Pot fi delimitate trei sensuri pentru termenul "**clasă**":

- O clasă defineşte ***scopul*** [intent] – structura şi comportamentul obiectelor de acelaşi tip. Scopul reprezintă tema, esența sau semantica unei clase.

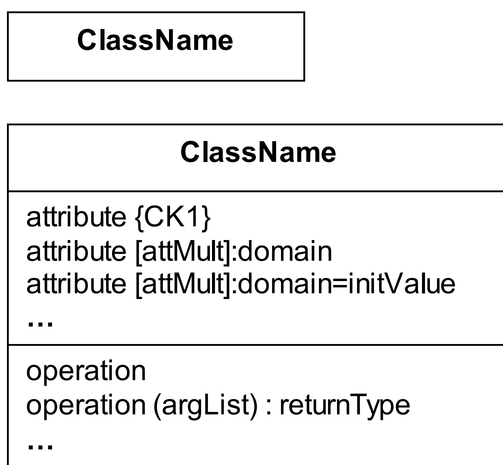
- O clasă defineşte un ***reprezentant*** [representative] – tipurile reprezentate de către obiecte. Acest sens [al termenului clasă] provine din programarea orientată-obiect în legătură cu abordarea bazată pe prototipizare sau pe delegare.
- O clasă defineşte o **extensie** [extent] – setul de obiecte de un anumit tip.

O ***responsabilitate*** reprezintă un contract sau o obligaţie [o îndatorire] a unei clase. La un nivel mai abstract, *atributele şi operaţiile corespunzătoare* [unei clase de obiecte] *sunt de fapt caracteristicile prin care sunt realizate responsabilităţile clasei*. Pe măsura rafinării modelelor, aceste responsabilităţi vor fi traduse într-un set de atribute şi operaţii care vor îndeplini cât mai bine responsabilităţile clasei.

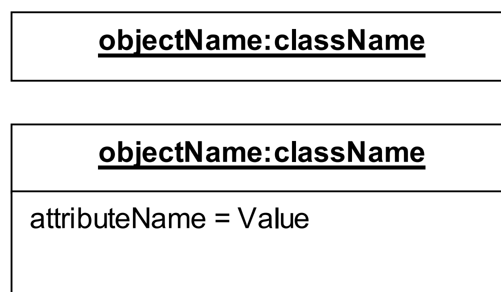
Un **atribut** reprezintă o proprietate [calificată prin nume – *named property*] care descrie un domeniu [range] de valori ce pot fi reţinute [stocate] de către instanţele respectivei proprietăţi. O ***valoare*** reprezintă un fragment dintr-o dată. Un ***atribut al unui obiect*** reprezintă o proprietate a unei clase care descrie o valoare păstrată de către fiecare obiect al clasei. Valorile şi obiectele nu trebuie confundate între ele. Obiectele au identitate, pe când valorile nu au.

O **operaţie** reprezintă implementarea unui serviciu care poate fi cerut de către orice obiect dintr-o clasă pentru a-şi manifesta comportamentul. În fapt, o **operaţie** reprezintă o funcţie sau o procedură care ar putea fi aplicată asupra sau de către obiectele unei clase. **O metodă** constituie implementarea unei operaţii pentru o clasă.

**Clasă:**




**Obiect:**




***Caracteristicile atributelor***

Formalizarea atributului unei clase se face după o notaţie specifică, cum ar fi:

- destinatar[*]: String = "o companie"

Astfel sunt sugerate caracteristicile unui atribut dintre care: *nume şi tip, vizibilitatea, multiplicitatea şi scopul*.

Unul dintre cele mai importante detalii care pot fi specificate pentru atributele şi operaţiile unui clase este **vizibilitatea**. Vizibilitatea unei caracteristici [feature] specifică dacă aceasta poate fi utilizată de către alţi clasificatori. Există trei nivele de vizibilitate în UML:

| | | |
|---|---|---|
| 1. | `public` | Orice clasificator exterior care are vizibilitate asupra clasificatorului dat poate folosi respectiva caracteristică; specificarea se face folosind simbol + ca prefix |
| 2. | `protected` | [protejat] Orice descendent al clasificatorului poate folosi această caracteristică; specificată prin prefixul # |
| 3. | `private` | [privat] Numai clasificatorul însuşi poate utiliza respectiva caracteristică; specificată prin prefixul – . |

***Multiplicitatea unui atribut*** specifică numărul de valori posibile pentru un atribut şi este precizat între paranteze pătrate după numele atributului. Poate fi specificată o valoare singulară obligatorie *[1]*, o valoare singulară opțională *[0..1]*, o colecție nespecificată având o limită inferioară *[limitaInf..*]*, sau o colecție cu limite fixe *[limitaInf..limitaSup]*. O limită inferioară zero permite valori nule, o limită inferioară unu sau multe interzice valorile nule.

*Multiplicitatea* reprezintă o caracteristică care se poate dovedi utilă şi în cazul unei *clase* pentru restricționarea numărului de instanțe pe care îl poate avea o clasă. Se pot specifica zero instanțe (caz în care este vorba despre o clasă utilitară [utility class] care prezintă doar atributele şi operațiile cu scop la nivel de clasă), o singură instanță (clase *singleton* sau unicat), un anumit număr de instanțe sau, cazul implicit, mai multe instanțe.

Un alt detaliu important pentru atributele sau operațiile un atribut sau operație este **scopul** posesorului său. Scopul unei caracteristiri [atribut sau operație] specifică dacă aceasta *apare în fiecare instanță* a clasei sau dacă există *doar o singură instanță* a caracteristicii pentru toate instanțele unui clase. O caracteristică care are ca scop *clasa* [sau la nivel de clasă] este reprezentată prin sublinierea numelui acesteia. Un ***atribut de clasă*** este un atribut a cărui valoare este comună unui grup de obiecte ale clasei, fiind mai puțin particularizabil pe fiecare instanță. Atributele de clasă pot fi utilizate pentru a păstra date implicite [default] sau rezumative pentru obiecte.

***Caracteristicile operațiilor***

Formalizarea atributului unei clase se face după o notație specifică, cum ar fi:

+ mărireSalariu(procM : Procent) : Numeric {leaf}

Astfel sunt sugerate caracteristicile unui atribut dintre care: *nume şi tip returnat, vizibilitatea (aceeaşi semnificație ca şi în cazul atributelor scopul* şi caracterul *abstract sau concret*.

O ***operație de clasă*** este o operație aplicabilă mai exact asupra clasei şi mai puțin asupra instanțelor clasei. Cel mai obişnuit gen de astfel de operații sunt cele de creare de noi instanțe [constructorii].

O operație este ***abstractă*** dacă există definită o interfață şi o funcționalitate, dar nu şi o metodă de implementare cu codul necesar. O ***operație abstractă*** specifică doar semnătura unei operații, amânând sau lăsând pe seama subclaselor derivate din clasa în care este definită implementarea.

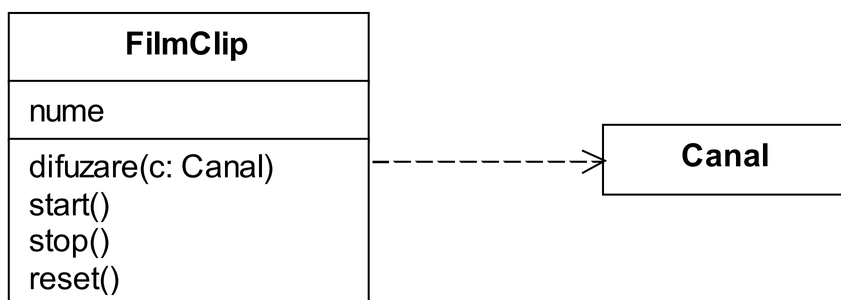
Caracterul abstract sau concret al operațiilor este strâns legat de *caracterul abstract sau concret al claselor* din care fac parte. Astfel anumite *clase* sunt desemnate *abstracte* însemnând că nu au nici un fel de instanțe directe. O clasă abstractă este indicată scriind numele ei italic. De

asemenea, se poate specifica şi faptul că o anumită clasă nu mai are la rândul ei copii [descendenţi]. Asemenea elemente sunt numite clase ***frunză*** fapt specificat în UML scriind proprietatea `leaf` sub numele clasei. Există şi posibilitatea specificării faptului că o clasă nu are părinţi [antecedenţi]. Un asemenea element este numit clasă ***rădăcină*** şi este specificat în UML scriind proprietatea `root` sub numele clasei. De asemenea o ***operaţie frunză*** (specificată prin proprietatea `leaf`) semnifică faptul că nu va putea fi redefinită în subclase, anulând astfel posibilitatea rescrierii polimorfice a acesteia.

**Relaţii**

*O* **relaţie** *constituie o conexiune [legătură] între entităţi.* În acest sens putem deosebi trei tipuri de relaţii:

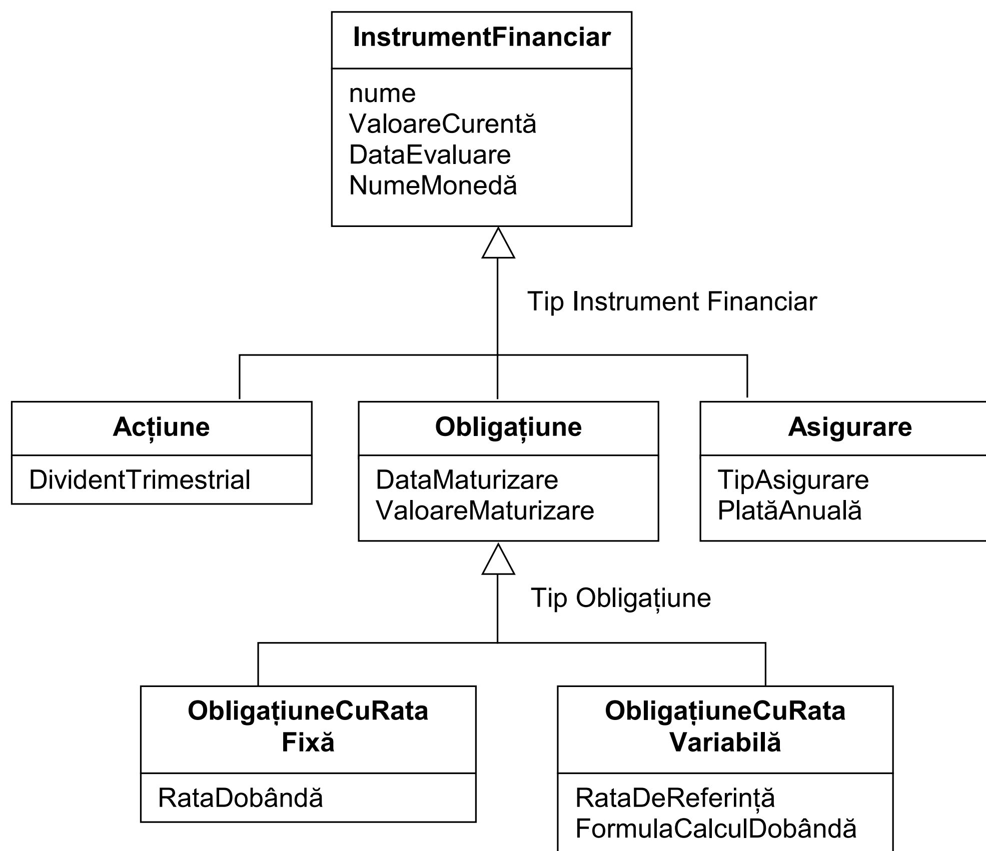
(a) ***Relaţii de dependenţă.*** O **dependenţă** [1]este o relaţie de utilizare [de întrebuinţare – using relationship] care stabileşte faptul că o schimbare în specificaţia unui lucru poate afecta [influenţa] un alt lucru care o utilizează, reciproca nefiind în mod necesar adevărată. Cel mai adesea dependenţele se folosesc pentru a arăta că o clasă utilizează o alta ca argument în semnătura [specificaţia] unei operaţii.




(b) ***Relaţii de generalizare.*** O **generalizare** este o relaţie între un element general (numit super-clasă sau părinte) şi un gen [kind] mai specific al acelui element (numit sub-clasă sau copil). Generalizarea este denumită adesea o relaţie "este un gen/tip de" [is-a-kind-of].

**Specializarea** furnizează o altă viziune asupra structurii sistemului. Specializarea are acelaşi înţeles ca şi generalizarea dar are o perspectivă de sus-în-jos, pornind de la superclasa pentru care sunt evidenţiate variantele acesteia. Un ***discriminator*** este un atribut care prezintă o valoare distinctă pentru fiecare subclasă; respectiva valoare indică care subclasă va descrie mai departe un obiect. Discriminatorul este pur şi simplu un nume pentru baza de generalizare.

[1] Nu înseamnă acelaşi lucru ca în ER (referire la dependeţa de existenţă)

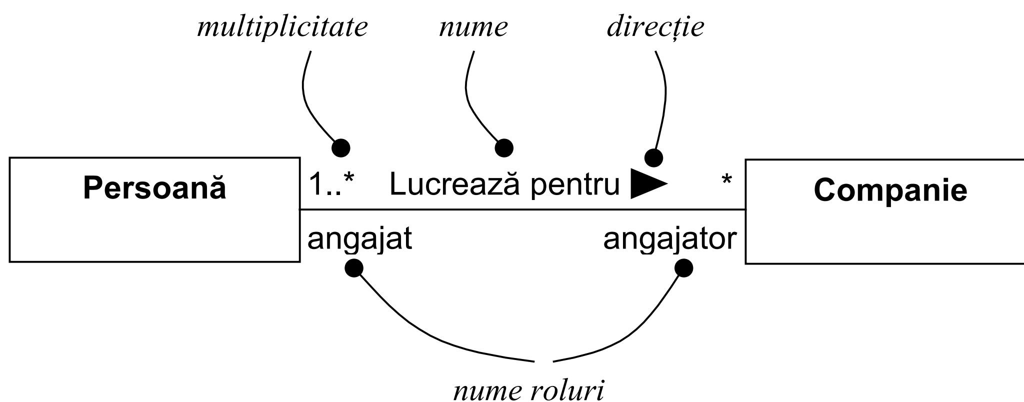



(c) ***Relații de asociere (asociații)***. O **asociere** [sau asociație] este o relație *structurală* care specifică faptul că obiectele unui lucru sunt conectate la [au legătură cu] obiectele unui alt lucru. Fiind dată o asociere care face legătura între două clase, se poate naviga de la obiectele unei clase la obiectele celeilalte clase, şi invers.

O ***legătură*** [link] reprezintă o conexiune fizică între obiecte. O ***asociație*** este descrierea unui grup de legături având o structură şi o semantică comune. Deci, o legătură este instanța unei asociații. O asociație descrie setul de legături potențiale în acelaşi mod în care o clasă descrie un set de obiecte de acelaşi tip.

O asociere poate avea un **nume** folosit pentru a descrie natura relației. Atunci când o clasă participă într-o relație ea are un **rol** specific pe care îl joacă în acea relație; un rol reprezintă "fațeta" pe care clasa de la capătul cel mai apropiat al asociației o prezintă clasei de la celălalt capăt.

Stabilirea a cât de multe obiecte pot fi conectate printr-o instanță a asociației înseamnă **multiplicitatea** rolului unei asociații. *Multiplicitatea* este o restricție asupra dimensiunii unei colecții şi este specificată prin *limita inferioară* şi *limita superioară* a numărului de elemente acceptat [1..1], [1..*], [0..*], [0..1], [2..4].

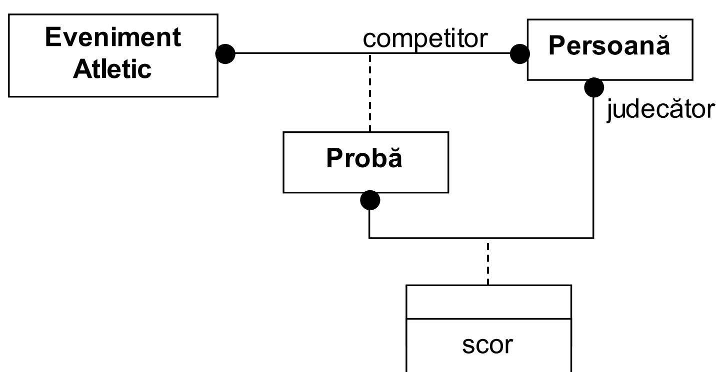



Dacă nu există alte specificații, **navigarea** printr-o **asociație** este bidirecțională. Există, însă, anumite situații în care se poate dori limitarea navigabilității într-o singură direcție. Fiind dată o asociație între două clase, obiectele unei clase pot vedea şi naviga către obiectele celeleilalte, dacă nu se restricționează navigabilitatea printr-o declarație explicită. În anumite cazuri se poate dovedi necesară limitarea **vizibilității** printr-o asociație relativ la obiectele din afara acesteia. În UML, se pot specifica trei nivele de vizibilitate pentru capătul unei asociații, la fel ca şi în cazul caracteristicilor claselor, adăugând la numele rolului un simbol cu referire la vizibilitate [ +, - sau # ]. Dacă nu există alte specificații, vizibilitatea unui rol este publică. Vizibilitatea privată indică faptul că obiectele de la acel capăt nu sunt accesibile nici unui obiect din afara asociației; vizibilitatea protejată indică faptul că obiectele de la acel capăt nu sunt accesebile oricăror alte obiecte din afara asociației cu excepția copiilor de la capătul celălalt.

***Compunerea*** – reprezintă o relație între un întreg (compozit) şi părțile sale (componente) reflectând faptul că nici compozitul nu poate exista fără părțile sale, nici componentele nu pot rezida în afara întregului (multiplicitate [1] la [1…*]). Exemple: DocumentComanda - ArticoleComanda

***Agregarea*** – reprezintă o relație între un întreg (agregat) şi părțile sale (constituente) reflectând faptul că agregatul ar putea exista fără părțile sale, iar părțile pot exista în afara întregului, pot aparține şi altor agregate. (multiplicitate [0..*] la [0…*]). Exemple: PlanInvațământ - Discipline

O ***clasă asociație*** este o asociație ale cărei legături pe lângă faptul că au atribute proprii ar putea participa şi în alte asociații.Ex: Persoana – *ContractAngajare* – Organizație, *ContractAngajare* – Departament - StructuraOrganizatorică.



**Diagrame de clase**

Fiind date conceptele de proiectare statică descrise mai sus, acestea sunt „asamblate” în diagrame de clase pentru a reprezenta [formaliza] un aspect particular dintr-o problemă sau sistem [software].

**O diagramă de clase** va modela un *mecanism*. Un mecanism reprezintă o funcţionalitate sau manifestare particulară a sistemului modelat care rezultă din interacţiunea unei comunităţi (societăţi) de clase, interfeţe şi alte elemente de acest gen. Se porneşte de la precizarea responsabilităţilor claselor care participă, din care se vor dezvolta ulterior atributele şi operaţiile.

Drept *definiţie* putem spune că o *diagramă de clase* este o diagramă care prezintă un set de clase, interfeţe şi colaborări precum şi relaţiile dintre ele.

De asemenea, diagramele de clase mai pot conţine package-uri şi subsisteme. Uneori este necesară şi includerea unor instanţe, în special în situaţiile în care se doreşte vizualizarea tipului (posibil dinamic) al unei instanţe.

### *2.2.2.2. Modelare comportamentală*

Modelarea comportamentală are în vedere aspectele privind modul în care obiectele *interacţionează* între ele şi modul în care acestea evoluează în timp sau cum îşi pot modifica starea curentă la un moment dat ca urmare a evenimentelor care declanşează tranziţiile de stare.

Conceptele care vor fi descrise în continuare se referă *la interacţiuni, obiecte, legături, mesaje* ce pot constitui subiectul *diagramelor de colaborare şi de secvenţe*.

**Interacţiuni şi legături**

O ***interacţiune*** reprezintă un comportament desemnat printr-un set de mesaje (inter)schimbate între membrii unui set de obiecte dintr-un context dat, pentru a realiza un anumit scop.

O ***legătură*** reprezintă o conexiune semantică între obiecte. În general o legătură este o instanţă a unei asociaţii. O legătură specifică o cale prin intermediul căreia un anume obiect trimite un mesaj către un alt obiect (sau către acelaşi obiect). De cele mai multe ori este suficient să se specifice că acea cale există. Prin urmare o *legătură* reprezintă calea prin care se asigură vizibilitatea între obiecte, iar tipurile standard sunt următoarele:

- **association** - pe baza unei asociaţii.
- **self -** specifică faptul că obiectul corespunzător este vizibil fiindcă el este expeditorul apelului
- **global** - specifică faptul că obiectul corespunzător este vizibil fiindcă aparţine contextului imediat superior [enclosing scope]
- **local** - specifică faptul că obiectul corespunzător este vizibil fiindcă se găseşte în spaţiul local [local scope]
- **parameter** - specifică faptul că obiectul corespunzător este vizibil fiindcă este un parametru

Un ***mesaj*** reprezintă specificaţia unei comunicări între obiecte care îşi transmit informaţii între ele aşteptând ca răspuns efectuarea anumitor activităţi. Recepţionarea unui mesaj poate fi

considerată instanța unui eveniment. *Atunci când este trimis un mesaj, acțiunea care rezultă este o comandă [instrucțiune] executabilă ce desemnează abstracțiunea unei proceduri*. O acțiune poate produce schimbarea stării obiectului. În UML pot fi modelate mai multe tipuri de acțiuni:

- `Apel` - invocă o operațiune pentru un obiect; un obiect își poate transmite singur un mesaj, rezultând o invocare locală a unei operații.
- `Return` - returnează o valoare obiectului apelant.
- `Send` - trimite un semnal unui obiect.
- `Create` - creează un obiect.
- `Destroy` - distruge un obiect; un obiect poate comite „suicid" dacă se distruge pe el insuşi.

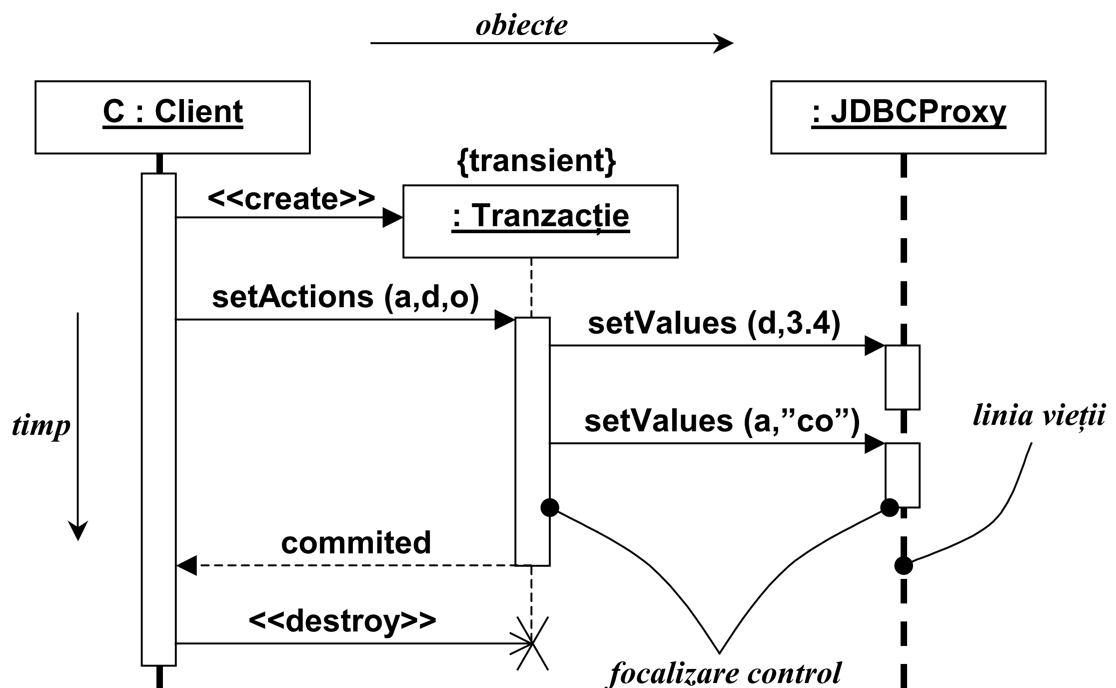
Atunci când un obiect apelează o operație sau transmite un semnal altui obiect, se pot furniza *parametri* care însoțesc mesajul respectiv. De asemenea, când un obiect returnează controlul altui obiect, poate fi reprezentată inclusiv *valoarea returnată*.

O **diagramă de interacțiune** prezintă o interacțiune, constând dintr-un set de obiecte şi relațiile dintre acestea, incluzând mesajele care ar putea fi schimbate între ele. În UML interacțiunile pot fi reprezentate sun formă a două tipuri de diagrame de interacțiuni: diagrame de secvențe şi diagrame de colaborare. **O *diagramă de secvențe*** este o diagramă de interacțiune care scoate în evidență ordinea în timp a mesajelor. **O *diagramă de colaborare*** este o diagramă de interacțiuni care scoate în evidență organizarea structurală a obiectelor care trimit şi recepționează mesaje.

**Diagramele de secvențe** au două caracteristici care le deosebesc față de diagramele de colaborare:

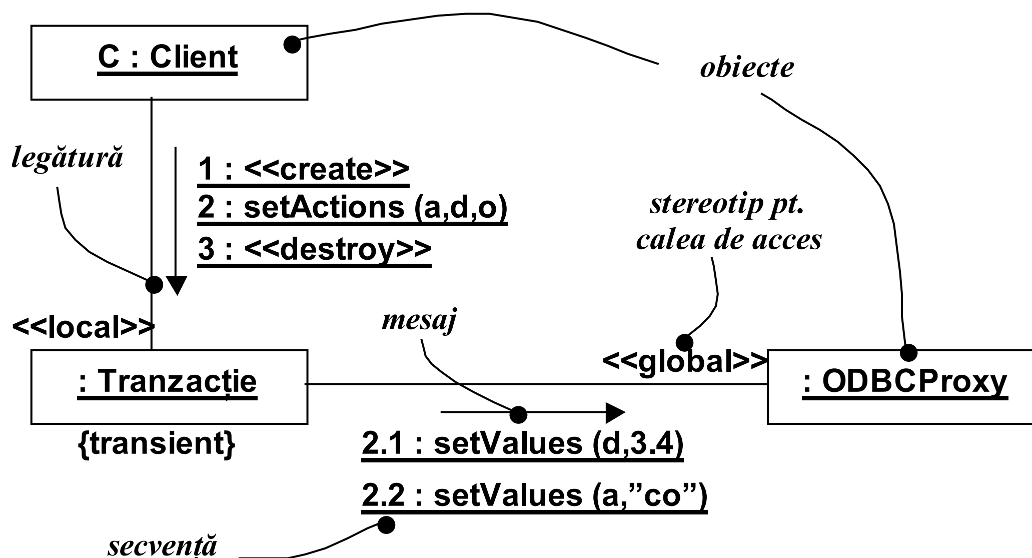
- În primul rând scot în evidență *linia vieții obiectului* [object lifeline]. O linie de viață pentru un obiect este o linie verticală întreruptă care reprezintă existența unui obiect într-o perioadă de timp. Majoritatea obiectelor care apar în diagramele de interacțiuni vor exista pe toată durata respectivei operațiuni, aşa încât obiectele sunt aliniate în partea superioară a diagramei, având liniile de viață trasate de sus în jos. Obiectele ar putea fi create în timpul interacțiunii caz în care liniile lor de viață încep odată cu recepționarea mesajului stereotipizat `create`. Obiectele ar putea fi distruse în timpul interacțiunii. Liniile lor de viață se încheie la primirea mesajului stereotipizat `destroy` (însoțit de o indicație vizuală sub forma unui `X` mare marcând sfârşitul existenței lor).
- În al doilea rând permit *focalizarea fluxului de control*. Focalizarea controlului este redată sub forma unui dreptunghi înalt şi subțire care arată perioada de timp în care obiectul efectuează o acțiune, în mod direct sau prin intermediul unei proceduri subordonate. Latura superioară a dreptunghiului este aliniată la momentul începerii acțiunii, latura inferioară este aliniată la momentul încheierii acțiunii (şi poate fi marcată prin returnarea unui mesaj). Îmbricarea focalizării controlului (datorată unei recursivități, apelului unei operații proprii sau unui callback al unui alt obiect) poate fi redată aşezând un alt dreptunghi pentru redarea focalizării controlului uşor la

dreapta părintelui său (lucru care poate fi repetat până atingerea unui subnivel arbitrar).




**Diagramele de colaborare** au la rândul lor două trăsături care le deosebesc de diagramele de secvențe.

- În primul rând scot în evidență *căile de acces*. Pentru a indica modul cum un obiect este legat de un altul se poate ataşa un indiciu (setereotip) indicând calea de acces, la capătul îndepărtat al legăturii (de ex. `<<local>>` indică faptul că obiectul desemnat este văzut ca local de către expeditor). De obicei legăturile au redată explicit calea pentru acces `local`, `parameter`, `global` şi `self` (dar nu şi pentru `association`).
- În al doilea rând fluxul de control este redat prin *numerele secvențial* ataşate mesajelor. Pentru a indica ordinea [succesiunea] în timp a unui mesaj, respectivul mesaj este prefixat cu un număr (pornind de la mesajul numerotat cu 1), incrementându-l uniform pentru fiecare mesaj nou din fluxul de control (2, 3 ş.a.m.d.). Pentru a reda îmbricarea, se foloseşte numerotarea zecimală Dewey (1 este primul mesaj, 1.1 este primul mesaj subinclus în mesajul 1, 1.2 este al doilea mesaj subinclus în mesajul 1.2 ş.a.m.d. până la atingerea unui nivel arbitrar). Se poate observa că de-a lungul unei singure legături pot fi redate mai multe mesaje (posibil provenind din direcții diferite), iar fiecare va avea un număr secvențial unic.




### 2.2.3. Implementarea structurală în Java a unui „business model" formalizat cu UML

Specificațiile grafice UML din diagramele de clase pot fi utilizate pentru generarea directă a structurile statice de implementare (definițiile claselor, atributelor, operațiilor, pachetelor).

Problema implementării modelelor formalizate în UML este destul de vastă, de aceea, pentru a simplifica acest complex de detalii, de vom limita la, să zice, clasele tipice din domeniul funcțional al aplicației (stratul logicii afacerii).

Regulile de implementare sunt dependente de atributele fiecărui element constructiv UML aşa cum este definit în meta-modelul UML. În acest sens ne vom limita să analizăm următoarele elemente: *clasele, atributele, operațiile, modulele* sau *package-urile* şi *rolurile asociațiilor.*

#### *2.2.3.1. Echivalența specificațiilor UML pentru clase, atribute, operații şi relații față de structurile constructive ale limbajului Java*

Cele mai importante caracteristici ale **claselor** UML cu incidență în implementare sunt:

- *final* sau *leaf* – clasă care nu mai prezintă subclase, va genera un modificator *final* în Java;
- *cardinalitatea 1* – clasă care prezintă o singură instanță în aplicație, va genera o clasă (eventual internă) în Java, în spațiul ei de rezidență fiind restricționată (programatic) la o singură instanță;
- *abstract* – clasă care nu implementează direct propriile instanțe, va genera o clasă *abstractă* în Java;
- *vizibilitatea:*

- *public* – generează în Java o clasă de sine stătătoare (fişier sursă propriu), însoțită de specificatorul `public`,
- *protected* generează în Java o clasă de internă, însoțită de specificatorul `protected`,
- *private* generează în Java o clasă de internă, însoțită de specificatorul `private`,
- *package or implementation* generează în Java o clasă de sine stătătoare, neînsoțită de specificatorul `public`, sau clasă definită în fişierul sursă al altei clase în definiția sau comportamentul căreia este implicată.

Cele mai importante caracteristici ale **atributelor** claselor UML cu incidență esențială în implementare sunt:

- *final – atributul* va fi specificat în Java, în definiția clasei din care face parte, prin cuvântul *final* specificând faptul că este vorba despre o constantă;
- *transient* - atributul va fi *specificat* în Java, în definiția clasei din care face parte, prin specificatorul *transient* (este ignorat de către operațiunile de serializare ale obiectelor în fişiere persistente);
- *vizibilitatea*
  - *Public (+)* - atributul va fi specificat în Java ca *public*
  - *Private (-)* - atributul va fi specificat în Java ca *private*
  - *Protected (#)* - atributul va fi specificat în Java ca *protected*
- *type* – numele *tipului* care va însoți în Java definiția atributului. Va specifica un tip specific mediului Java (limbajul – clasele fundamentale, bibliotecile suport – clasele arhitecturale sau clase specifice domeniului afacerii sau unei aplicații particulare).

Cele mai importante caracteristici ale **operațiilor** claselor UML cu incidență esențială în implementare sunt:

- *abstract* – în *definiția* operației din clasa Java va fi inclus specificatorul *abstract* simbolizând faptul că este vorba despre o operație fără implementare dintr-o clasă abstractă,
- *static* - în definiția *operației* din clasa Java va fi inclus specificatorul *static* simbolizând faptul că este vorba despre o operație definită la nivelul clasei şi care nu depinde de fiecare instanță în parte,
- *final* în definiția *operației* din clasa Java va fi inclus specificatorul *static* simbolizând faptul că această operație nu va fi suprascrisă în subclase,
- *argumentele* - în definiția *operației* din clasa Java va fi inclus câte un parametru cu numele şi tipul specificat în definiția operației clasei din modelul UML,
- *type* – în *definiția* operației din clasa Java va fi inclus ca tip returnat,
- *vizibilitatea*
  - *public (+)* - operația va fi specificată în Java ca *public,*
  - *private (-)* - operația va fi specificată în Java ca *private,*
  - *protected (#)* - operația va fi specificată în Java ca *protected.*

**Rolurile** specificate „capetelor” asociațiilor din diagramele UML sunt implementate în Java în general ca atribute de sine stătătoare. Principalele caracteristici ale **rolurilor** claselor UML în asociații cu incidență esențială în proiectare sunt asemănătoare atributelor. Problemele mai deosebite în legătură cu asociațiile au în vedere:

- *navigabilitatea* asociațiilor:
  - asociațiile *bidirecționale* vor genera câte un atribut în fiecare clasă implicată în asociație;
  - asociațiile *unidirecționale* vor genera un singur atribut la capătul simplu al asociației (capătul care desemnează punctul inițial al direcției);
- *multiplicitatea* asociațiilor
  - într-o asociație *unu-la-multe*, capătul *multe* va fi implementat printr-o colecție (de regulă un *array*);
  - într-o asociație *multe-la-multe*, ambele capete vor trebui implementate printr-o colecție;
- *tipul* rolurilor – rolurile ar putea fi implementate prin *interfețe* specifice în Java, caz în care un rol va determina crearea unei clase *abstracte* sau a unei componente *interface*, ale căror tipuri vor însoți definiția atributului care va implementa rolul în clasa implicată în asociație.

În UML mai există însă două tipuri de relații:

- *dependențe* – care nu vor avea un tratament specific în cazul implementării în Java (sunt reflectate pur şi simplu prin faptul că anumite argumente sau variabile interne ale unor metode au ca tip declarat o anume clasă distinctă);
- *generalizări* – care vor conduce la ierarhii de moştenire în Java, *subclasele* fiind legate prin cuvântul cheie *extends* de superclase. Mai trebui făcută menţiunea că în Java nu este posibilă moştenirea multiplă, ca urmare modelele UML care implică moştenire multiplă vor trebui reformulate pentru a include doar moştenire simplă. Singurul caz particular ar moştenirii multiple care poate fi implementat în Java se referă la implementarea (cuvântul cheie *implements*) unor *interfețe* de către aceeaşi clasă.

#### *2.2.3.2. Cazuri punctuale de „translatare” a specificațiilor UML din diagramele de clase în limbajul Java*

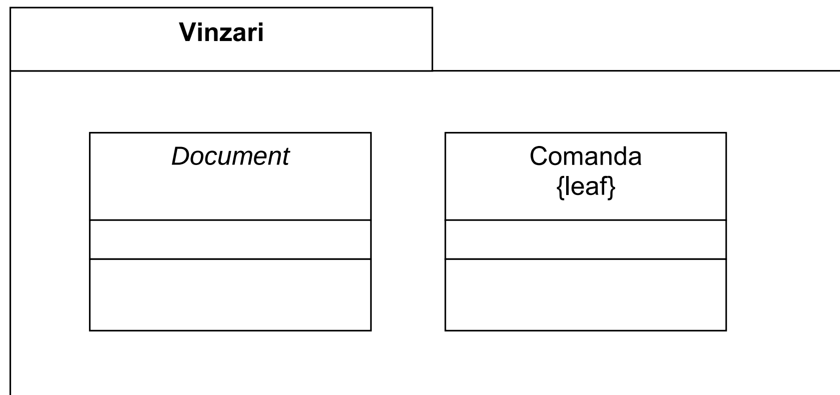
În continuare vom exemplifica traducerea fiecare element constructiv UML esențial în constructorii unui limbaj OO – Java.

**Clasele UML**

După cum am arătat în paragraful anterior, elementele esențiale care vor rezulta prin „traducerea” acestora într-un limbaj de programare OO (Java) se referă la numele clasei, vizibilitatea clasei, specificația (semnătura) contructorului default al clasei, atributele, operațiile, atributele-referință semnificând relațiile asociații.

De asemenea, abstractizarea modulelor sau subsistemelor se face prin intermediul *package*-urilor care reprezintă mecanismul de grupare obişnuit în UML. În mod transparent, pachetele UML pot fi traduse în pachetele specifice Java, păstrând relaţiile de includere între ele (traduse prin foldere şi subfoldere) şi de includere a claselor sau interfeţelor care formează conţinutul lor prin intermediul cuvântului cheie corespunzător în Java, şi anume ***package***.

De exemplu, următoarele specificaţii vizuale



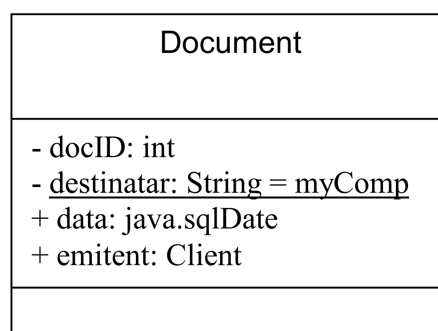

ar trebui interpretate în Java astfel:

```
----------- fisierul sursă Document.java ---------------------------
package vinzari;
public abstract class Document {
        public Document() {
    }
}
----------- fisierul Comanda.java ---------------------------
package vinzari;
public final class Comanda {
    public Comanda() {
    }
}
```

**Atributele şi operaţiile**

În legătură strict cu ***atributele***, care fac parte din definiţiile claselor, elementele esenţiale care pot rezulta din specificaţiile UML se referă la: *tip,vizibiliatate, scop static, nume*.

De exemplu, următoarele specificaţii vizuale




pot fi traduse în:

```
public class Document {
    private int docID;
    private static String destinatar = "myComp";
```

```
        public java.sql.Date data;
        public Client emitent;
        /** Creates a new instance of Document */
        public Document() {
        }
    }
```

***Operațiile*** care fac parte din definiția claselor, vor fi traduse în Java ținând cont de următoarele elemente deduse din „semnăturile” UML: *vizibilitate, scop static, tip (returnat), nume, argumente*.

Astfel, următoarele specificații UML

| Document |
|---|
| - docID: int<br>- destinatar: String = myComp<br>+ data: java.sqlDate<br>+ emitent: Client |
| - genereazaDocID() : int<br>+ getEmitent() : Client<br>+ setDataDoc(dataDoc) {leaf} |

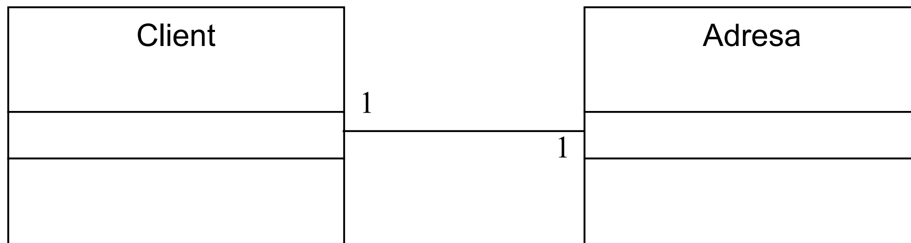
vor fi traduse în Java prin:

```
    public class Document {
        private int docID;
        private static String destinatar = "myComp";
        public java.sql.Date data;
        public Client emitent;
        /** Creates a new instance of Document */
        public Document() {
        }
        private int genereazaDocID(){
            return 0;
        }
        public Client getEmitent(){
            return null;
        }
        public final void setDataDoc(java.sql.Date dataDoc){
        }
    }
```

**Relațiile de asociere și generalizare**

Relațiile care au caracteristici deosebite pentru limbajul de implementare sunt, după cum am văzut mai înainte, *asociațiile* și *generalizările*.

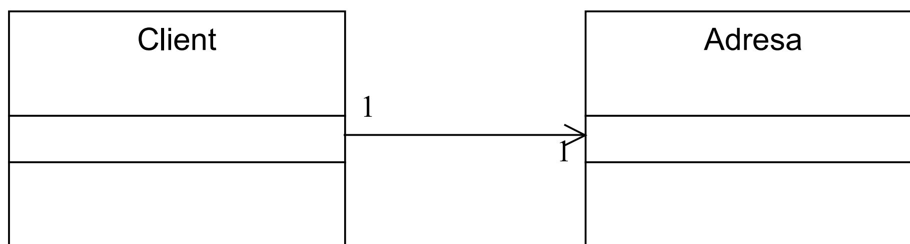
**Asociațiile bidirecționale** apar de regulă în UML astfel:




în Java, apărând următoarele specificații:

```
public class Client{
   public Adresa sediu;
   public Client(){}
}
---------------------------------------------
public class Adresa {
   public Client rezident;
   public Adresa(){}
}
```

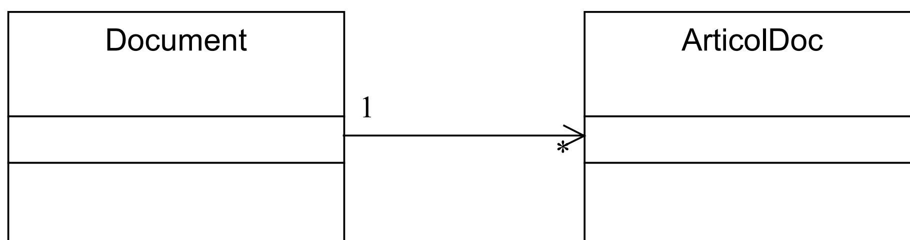
Dacă asociația ar fi fost *unidirecțională*




atunci clasa către care punctează direcția de navigare n-ar mai păstra o referință către clasa origine a direcției de navigare:

```
public class Client{
   public Adresa sediu;
   public Client(){}
}
---------------------------------------------
public class Adresa {
   public Adresa(){}
}
```

***Asociațiile una-la-multe*** sunt formalizate de regulă în UML astfel:

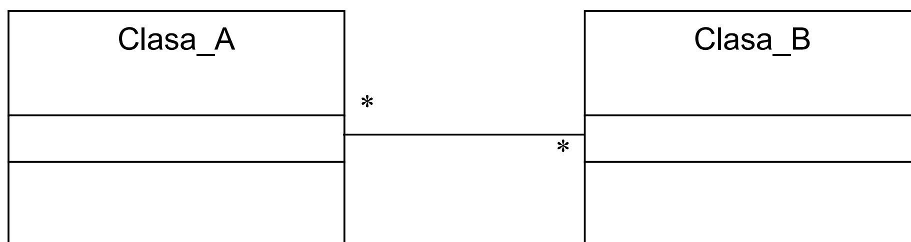



în Java fiind traduse prin:

```
public class Document {
   public ArticolDoc[ ] detalii;
   ... ... ...
```

```
    /** Creates a new instance of Document */
    public Document() {
    }
}
------------------------------------------------
public class ArticolDoc{
    public ArticolDoc(){}
}
```

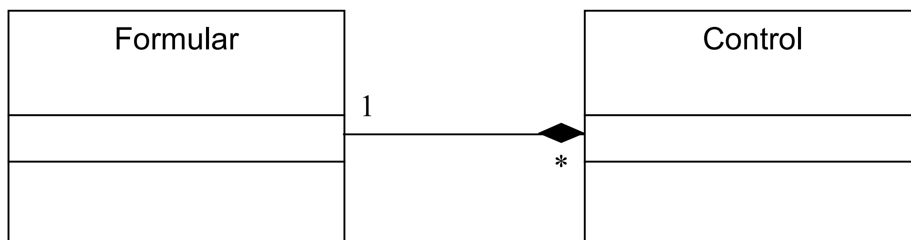
*Asociațiile multe-la-multe* apar în UML astfel:




în Java rezultând următoarea structură:

```
public class Clasa_A {
    public Clasa_B[ ] rel_cls_B;
    ... ... ...
    public Clasa_A() {}
}
------------------------------------------------
public class Clasa_B{
    public Clasa_A[ ] rel_cls_A;
    public Clasa_B(){}
}
```

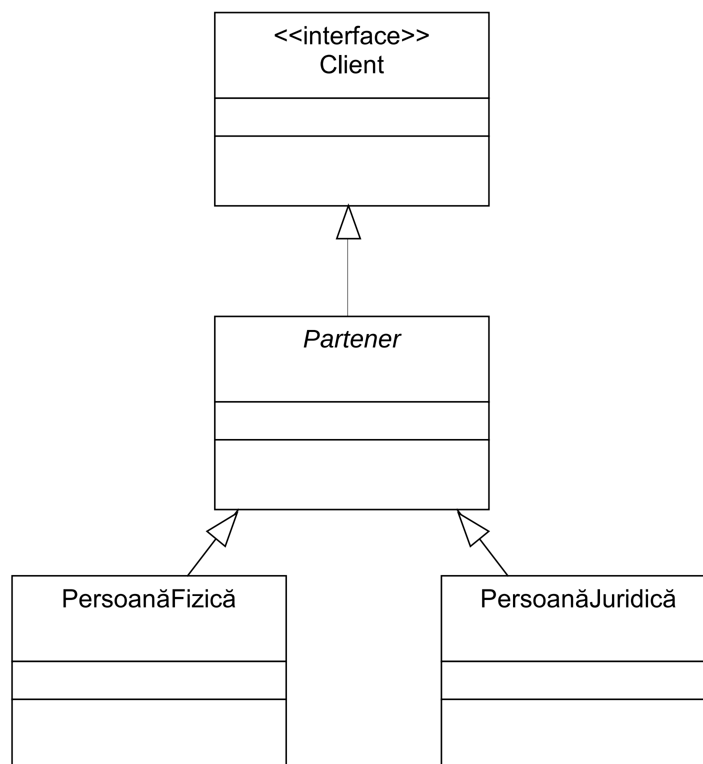
*Relațiile de agregare* sunt reprezentate în UML folosind simbolul unui romb:




în Java apărând ca o relație obişnuită unu-la-multe sau unu-la-unu. În cazul în care este vorba de multiplicitate „multe” în capătul „părții” atunci, opțional, se poate recurge la definirea întregului ca fiind (sau derivând) dintr-un *Container*.

```
public class Formular extends Container{
    public Component[ ] componenteGrafice;
    ... ... ...
    public Formular() {}
}
------------------------------------------------
public class Control extends Component{
    public Container formular;
    public Control(){}
}
```

***Relațiile de generalizare*** din UML se traduc prin *extends* (părintele se referă la o clasă simplă sau abstractă) sau *implements* (părintele este o interfață).




Specificațiile UML de mai sus, le putem implementa în Java astfel:

```
public interface Client{
   String getNumeClient();
}
---------------------------------------------------------------
public abstract class Partener implements Client{
   public Partener() {
   }
   public abstract String getNumeClient();
}
---------------------------------------------------------------
public class PersoanaFizica extends Partener{
   public String getNumeClient() {
      return null;
   }
}
---------------------------------------------------------------
public class PersoanaJuridica extends Partener{
   public String getNumeClient() {
      return null;
   }
}
```

### 2.2.4. Implementarea relațională a unui „business model” formalizat cu UML. Principii privind maparea obiectual/relațional

*Clasele entități* sunt acele clase ale căror instanțe sunt stereotipizate ca *persistent*, adică timpul de viață al acestora depăşeşte eventual timpul de viață al aplicațiilor în care sunt invocate şi prin urmare au nevoie de un spațiu de rezidență de genul unei baze de date.

Implementarea *claselor entități* este echivalentă de fapt cu dezvoltarea sau construirea bazei de date corespunzătoare modelului aplicației dezvoltate. În acest scop trebuie parcurse mai multe etape:

- **Implementarea identității** – traducerea în constructorii specifice SGBD-urilor relaționale a formei de exprimare a identități tabelelor
- **Implementarea domeniilor**
- **Implementarea claselor -** definirea tabelelor ale căror linii vor reflecta instanţele claselor entități şi/sau asociațiilor dintre clasele de entități (în particular clasele-asociații).

#### *2.2.4.1. Implementare identității*

Abordările cele mai des invocate în acest sens sunt identitatea bazată pe existență şi identitatea bazată pe valoare.

**Identitatea bazată pe existență**

Această abordare presupune adăugarea unui atribut OID (identificator de obiect) fiecărei tabele corespunzătoare unei clase de entități şi marcat drept cheie primară. Cheia primară pentru fiecare tabelă de asociații constă din identificatorii uneia sau mai multor clase legate [prin acea asociație]. În mod ideal ar trebui utilizată aceeaşi valoare de identificare pentru un obiect de-a lungul întregii ierarhii de moştenire. Identificatorii nu sunt obligatoriu reprezentați în modelul obiect, însă trebuie incluşi în schema tabelei.

Cea mai eficientă tehnică în acest sens prespune folosirea unei coloane a cărei unicitate este absolut independentă de valorile celorlalte attribute. Spre exemplu, în Oracle, ideală în acest sens este *pseudo*-coloana **ROWID**.

**Identitatea bazată pe valoare**

Această abordare presupune folosirea unui atribut scalar (sau a unui set de atribute scalare) ale cărui valori (combinație de valori) să reflecte unicitatea obiectelor şi care să constituie baza definirii cheii primare în tabela relațională care reflectă clasa entitate.

În acest sens putem întâlni mai multe situații

- **Clase independente**. Fiecare clasă dispune de un atribut/set de atribute care vor genera complet cheia primară.
- **Clase dependente prin asociații**. Obiectele unei *clase dependente* îşi derivă identitatea din alte obiecte. Prin urmare cheia primară în tabela care reflectă această clasă va prelua, pe lângă atributele desemnate din clasa sursă, şi atributele de identitate din clasele de care depinde clasa entitate.

- **Clase dependente prin generalizare**. În mod normal cheia primară a superclasei se propagă subclaselor.

#### 2.2.4.2. *Implementarea domeniilor*

*Domeniile* reprezintă seturile de valori din care vor fi selectate valorile individuale ale atributelor claselor. În acest sens putem întâlni următoarele situaţii:

**Domeniul identificatorului.** Multe SGBDR-uri facilitează identitatea bazată pe existenţă prin intermediul domeniiilor de identificator. De exemplu se pot defini [obiecte] secvenţe în Oracle (*CREATE SEQUENCE sequenceName*). Atunci când se inserează o înregistrare, se specifică numele secvenţei ca valoare iar Oracle furnizează următorul număr întreg.

**Domeniile enumerative [colecţii]**. Un domeniu enumerativ restricţionează un atribut la un set (colecţie) de elemente (valori) de acelaşi tip. Implementarea domeniilor enumerative poate fi gândită în mai multe moduri, dintre care:

- **Şir enumerativ** [enumeration string]. În această abordare se stochează pur şi simplu un atribut enumerativ ca un şir [de caractere]. Dacă SGBDR-ul permite se poate defini o restricţie care să limiteze enumerarea doar la valori permise.
- **Un flag per fiecare valoare din enumerare.** Se poate defini un atribut boolean pentru fiecare valoare din enumerare.

**Domeniile structurate.** Domeniile structurate pot fi expandate în domenii mai mici care la rândul lor pot fi structurate sau simple. În figura următoare sunt prezentate trei căi pentru implementarea domeniilor structurate.

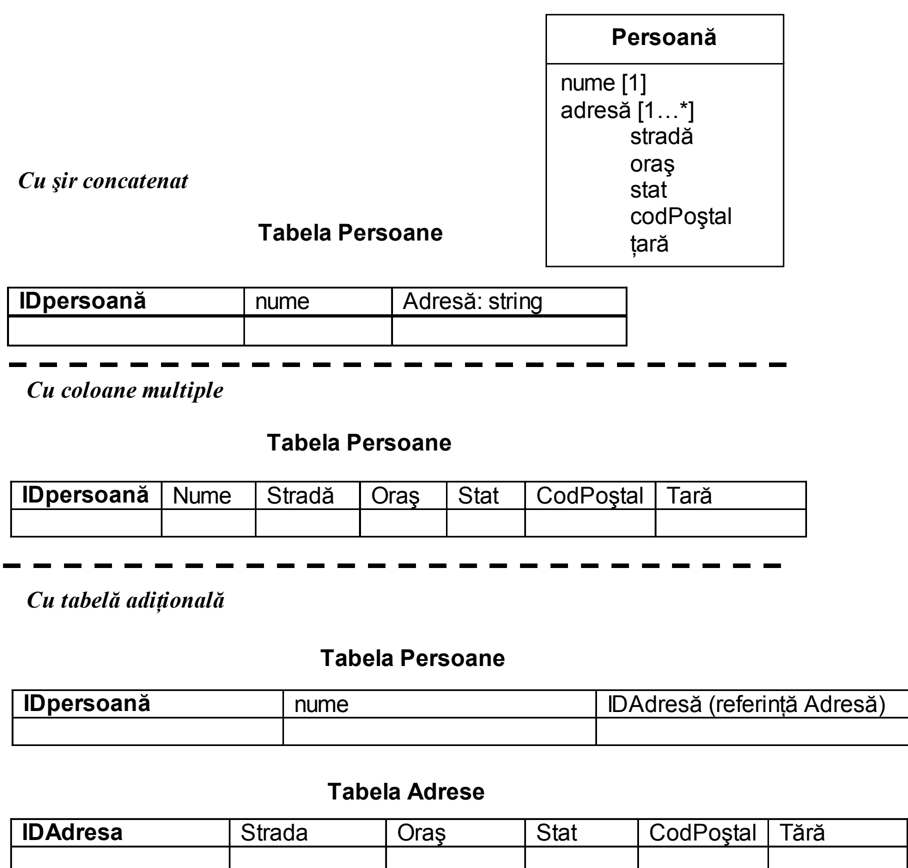



**Tabela Persoane**

| **IDpersoană** | nume | Adresă: string |
|---|---|---|
| | | |

**Tabela Persoane**

| **IDpersoană** | Nume | Stradă | Oraş | Stat | CodPoştal | Tară |
|---|---|---|---|---|---|---|
| | | | | | | |

**Tabela Persoane**

| **IDpersoană** | nume | IDAdresă (referință Adresă) |
|---|---|---|
| | | |

**Tabela Adrese**

| **IDAdresa** | Strada | Oraş | Stat | CodPoştal | Tără |
|---|---|---|---|---|---|
| | | | | | |

Figura - Abordări privind implementarea domeniilor structurate, după [BLAHA&PREMERLANI98, 281]

**Domeniile multivaloare**. Muliplicitatea atributelor se referă la numărul de valori pentru un atribut. Pot fi specificate o valoare singulară obligatorie *[1]*, o valoarea singulară opțională *[0..1]*, o colecție cu precizarea limitei inferioare *[limitaInferioară..*]* sau o colecție cu limite fixe *[limitaInferioară…limitaSuperioară]*. SGBR-urile nu implementează domenii multivaloare [puține pemit aşa-zisele tabele îmbricate – nested tables]. Tehnicile folosite pentru implementarea domeniilor structurate ar trebui utilizate şi pentru implementarea domeniilor multivaloare (concatenare, coloane multiple sau tabele adiționale).

### *2.2.4.3. Implementarea claselor şi relațiilor*

În mod normal **fiecare clasă se mapează într-o tabelă separată**, în care fiecare atribut este o coloană. Se cer coloane adiționale pentru:

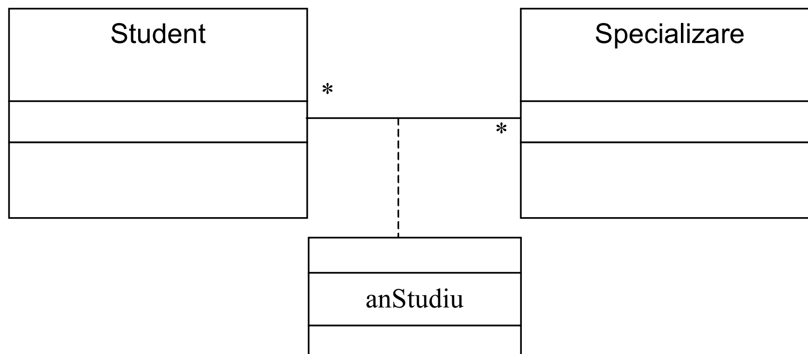
- identificatorul generat (identitatea bazată pe existență),
- descompunerea domeniilor enumerative în atribute booleene
- descompunerea atributelor structurate

Probleme mai deosebite apar în legătură cu asociațiile dintre clase (inclusiv clasele-asociații) şi generalizărilor.

**Implementarea asociațiilor simple**

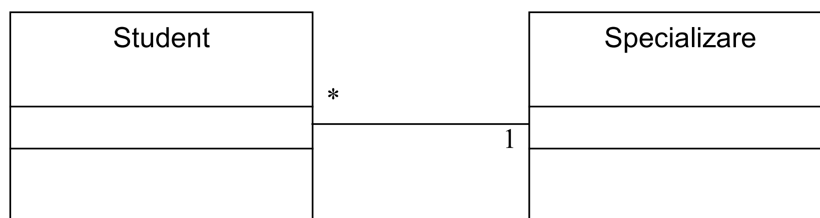
Dintre tehnicile cele mai "populare" în această privință putem aminti:

- **Tabelă distinctă pentru asociațiile multe-la-multe**. Fiecare asociație multe-la-multe ar trebui mapată într-o tabelă distinctă. Cheia primară a acestei asociații va fi combinarea cheilor primare din fiecare tabelă.




**Studenti**{<u>matricol,</u> **...**}
**Specializare**{<u>idSpec,</u> **...**}
**StudSpec**{<u>matricol</u>**,** <u>idSpec</u>, anStudiu }

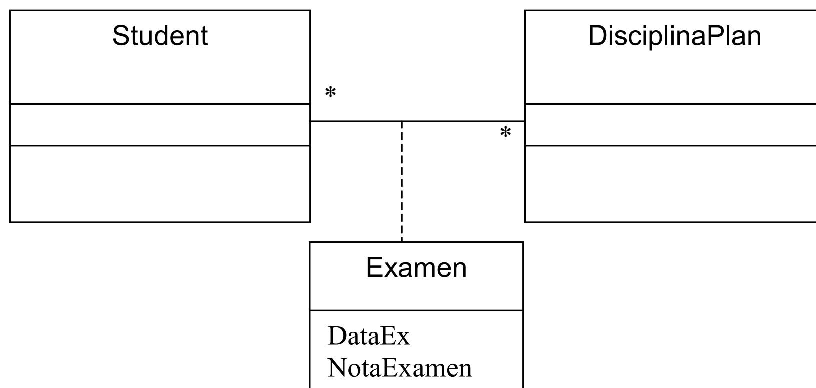
- **Asociațiile unu-la-multe incluse într-o tabelă corespunzătoare unei clase din relație**. O asociație unu-la-multe poate fi implementată printr-o cheie străină inclusă în tabela clasei având rolul "multe".




**Studenti**{<u>matricol,</u> **idSpec,...**}
**Specializare**{<u>idSpec,</u> **...**}

- **Asociații unu-la-unu**. Cheia străină poate fi definită în oricare din tabelele claselor.
- **Tabelă distinctă**. Asociațiile unu-la-multe sau unu-la-unu pot fi, de asemenea, implementate ca tabele distincte. Pentru asociația unu-la-multe se alege cheia străină pentru ramura "multe" ca şi cheie primară pentru tabela asociație. Pentru o asociație unu-la-unu oricare din ele poate fi aleasă.

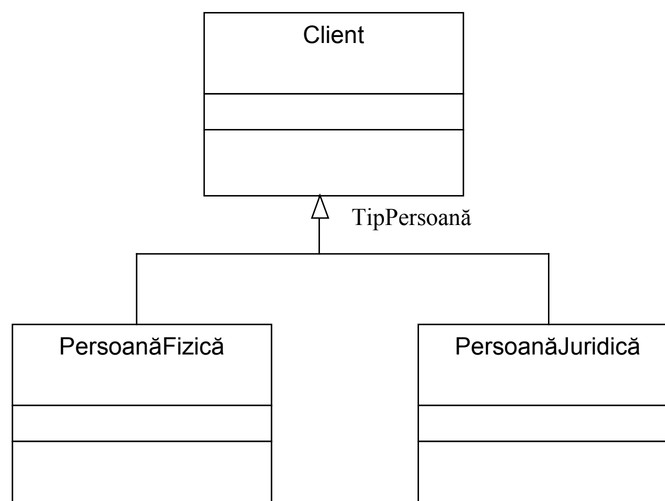
Ca regulă, **clasele de asociații** se implementează ca tabele distincte. Instanțele claselor de asociații primesc identitatea propagată de la clasele legate [prin asociație].




**Studenti**{<u>matricol</u>, … …}
**DisciplinaPlan**{<u>codDisc</u>, <u>codSpec</u>, … …}
**Examene**{<u>matricol</u>, <u>codDisc</u>, <u>codSpec</u>, DataEx, NotaExamen}

**Implementarea moştenirii simple**

*Generalizarea* se implementează de multe ori prin tabele separate pentru superclasă şi subclase. În mod ideal un obiect ar trebui să aibă aceeaşi cheie primară de-a lungul ierarhiei de moştenire. În tabele superclasei se va reflecta şi *discrimanatorul* care va indica tabela subclasă corespunzătoare pentru fiecare înregistrare din superclasă. Pentru o generalizare pe mai multe nivele ar trebui aplicată maparea pe rând pentru fiecare nivel în parte. Integritatea referenţială va asigura ca fiecărei înregistrări din subclasă să-i corespundă unei înregistrare din superclasă. De asemenea, poate fi definit un *view* pentru fiecare subclasă pentru a consolida moştenirea datelor şi a uşura accesul la obiecte.




**Clienti**{<u>id</u>, Nume, TipPersoana, …, …}
**PersoaneFizice**{<u>CNP</u>, *id_Client*, Adresa, … ...} – id_client este cheie străină pe restricţia referenţială cu Clienţi
**PersoaneJuridice**{<u>CodFiscal</u>, *id_client*, Sediu, TipScocietate, … …} – id_Client este cheie străină pe restricţia referenţială cu Clienţi

Alte tehnici de mapare alternative, cu anumite dezavantaje, sunt următoarele:

- **Eliminarea**. Tabele care nu au alte atribute decât cheia primară pot fi eliminate, înregistrările subclasei respective fiind incluse în tabela superclasei.
- **“Împingerea” în jos a atributelor superclasei**. Generalizarea poate fi de asemenea implementată prin eliminarea tabelei superclasei şi replicarea atributelor superclasei în fiecare subclasă. Un obiect poate fi astfel descris într-o singură tabelă, în locul raspândirii descrierii printre alte tabele pentru fiecare nivel al ierarhiei.
- **“Împingerea” în sus a atributelor subclaselor**. O altă opțiune este “împingerea” atributelor către superclasă şi eliminarea tabelelor subclaselor (o singură tabelă pentru o ierarhie de clase). Atributele subclasei care nu se aplică unui obiect dat sunt stabilite ca null-e.

**Clienti**{id, Nume, TipPersoana, CNP, CodFiscal, sediu, adresa, tipSocietate…}

- *tipPersoana* va determina de fapt clasa specifică (CHECK pentru ‘Persoană Fizică’ şi ‘Persoană Juridică’)
- câmpurile *CNP*, *adresa* este restricționate să aibă valori doar pentru *tipPersoană = ‘Persoană Fizică’*;
- câmpurile *CodFiscal, sediu* şi *tipSocietate* sunt restricționate să aibă valori doar pentru *tipPersoană = ‘Persoană Juridică’*;

- **Abordarea hibridă**. Se pot “împinge” atributele superclasei în jos în ierarhie şi se păstrează tabela superclasei pentru navigarea în subclase.
- **Tabela de generalizare**. O ultimă alternativă este folosirea unei tabele superclase, mai multor tabele subclase şi a unei tabele de generalizare la care se adaugă tabela de generalizare ce va lega cheia primară a superclasei de cheile primare ale subclaselor.

**Clienti**{id, Nume, TipPersoana, …, …}
**PersoaneFizice**{id persoana, CNP, Adresa, … ...}
**PersoaneJuridice**{id persoana, CodFiscal, Sediu, TipScocietate, … …}
**ClientiPersoane**{id client, id persoana, tipPersoana}

## 2.3. Asigurarea serviciului de persistență pentru clasele din domeniul afacerii. JDBC şi tehnica DAO (data access objects)

O *definiție* corectă şi concisă a persistenței poate fi considerată următoarea: „un *obiect persistent* reprezintă un obiect care continuă să existe şi după încheierea timpului de execuție al programului care manipulează acel obiect”[2] [Budd2002, 579].

---

[2] Budd, Timothy *An introduction to object-oriented programming Third Edition*, Addison Wesley, Pearson Education, Inc., 2002, pagina 579

Prin urmare persistenţa ar fi de fapt capacitatea de a salva starea obiectelor pe un suport permanent şi de a le reconstitui în mod transparent în spaţiul de execuţie al aplicaţiilor.

Mediile de programare obiectuale oferă implicit facilităţi de obţinere a persistenţei printr-un proces numit *serializarea obiectelor*. În Java, spre exemplu, acest proces se realizează în principal marcând aceste obiecte ca fiind serializabile prin implementarea de către clasele acestora a unei interfeţe specifice *java.io.Serializable*. Permanentizarea acestor obiecte într-un fişier este însă delegată unor clase specifice separate *ObjectOutputStream* (scrierea) şi *ObjectInputStream* (citirea). Prin urmare lucrul efectiv cu fişierul sau structura de stocare este delegată altor clase care ascund detaliile fizice. Pentru aplicaţiile complexe de întreprindere în medii multi-utilizator, concurente, distribuite, accesate de la distanţă acest model de persistenţă este total nesatisfăcător datorită lipsei mijloacelor implicite pentru gestionarea concurenţei, tranzacţiilor etc. Acest tip de facilităţi sunt asigurate la cel mai bun nivel de performanţă prin serverele de baze de date, iar „partea leului” în acest sens o deţin bazele de date relaţionale.

Prin urmare, din motive practice mai mult decât teoretice, calea de urmat ar fi să se încerce a se asigura persistenţa obiectelor într-o structură sau mediu de stocare relaţional, problema cheie fiind ca acest lucru să se facă într-o manieră la fel de transparentă ca şi seriabilizarea obiectelor în fişiere obişnuite. Problema ar putea fi sintetizată în obţinerea unei ***seriabilizări relaţionale*** a obiectelor gestionate în medii obişnuite orientate-obiect. Pentru a atinge acest obiectiv suntem nevoiţi, pe de o parte, să cunoaştem detaliile modulului de acces la structurile de stocare ale unei baze de date şi, pe de altă parte, să luăm în considerare o tehnică specifică pentru abstractizarea nivelului de persistenţă astfel încât aplicaţia în sine să fie relativ independentă faţă de detaliile brute ale accesului SQL.

#### 2.3.2.1. Acces universal la structurile de stocare specifice bazelor de date: biblioteca JDBC

**JDBC** asigură o interfaţă (API) standard pentru accesarea bazelor de date relaţionale din aplicaţii Java indiferent de locaţia în care se găseşte serverul bazei de date sau locaţia în care se găseşte distribuită baza de date. Cu alte cuvinte JDBC oferă o cale prin care fraze SQL (corespunzătoare pe cât posibil standardului ANSI-SQL recunoscut şi mai puţin clonelor SQL implementate în platformele de baze de date particulare) sunt transmise pentru a fi executate de către SGBD-ul gazdă.

Structura unei aplicaţii folosind tehnologia JDBC pentru conectarea la baza de date arată ca în figura următoare.

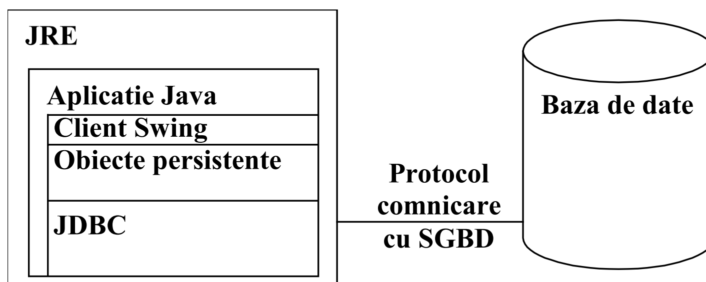



Figura - Arhitectura unei aplicaţii care accesează o bază de date prin JDBC

Se poate spune că JDBC reprezintă o sumă specificații prin care programatorul aplicației Java este scutit de sarcina codificării accesului la un SGBD proprietar, fiecare producător al unui SGBD va furniza un driver specific care va respecta interfața JDBC aşa încât aplicația Java să poată accesa datele într-o manieră (cât mai) portabilă față de bazele de date (relaționale) proprietare.

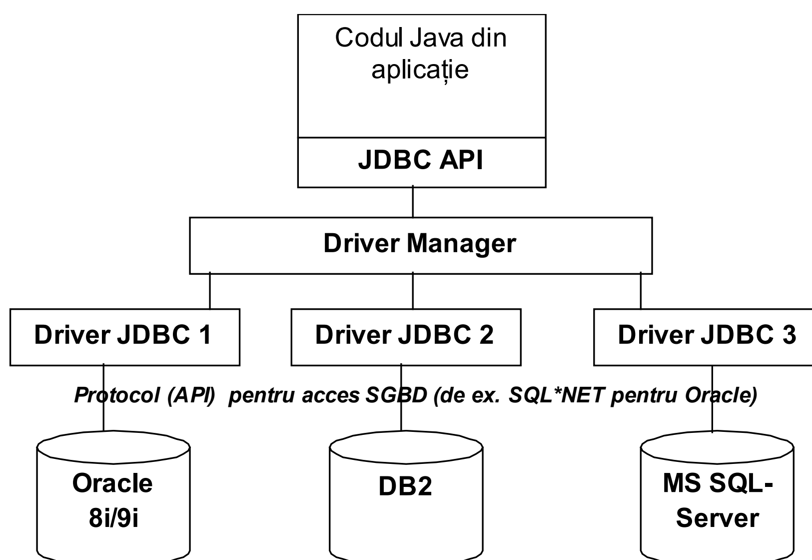



Figura - Relația dintre API JDBC şi mecanismul de comunicare cu bazele de date

JDBC API oferă astfel cale de a utiliza serviciile esențiale ale SGBD-ului folosit pentru stocarea datelor persistente din domeniul aplicației, şi anume:

1. Stabilirea conexiunii la baza de date (invocând bineînțeles mecanismul de securitate al SGBD-ului);
2. Lucrul cu structurile logice relaționale (tabele, view-uri etc.) pentru stocarea/regăsirea datelor prin fraze conforme standardului SQL;
3. Administrarea mecanismului tranzacțional oferit de SGBD (fie automat de către driver-ul JDBC, fie manual în aplicație);
4. Serviciile de partajare sau concurență la nivelul datelor prin mecanismele de blocare şi seriabilizare a tranzacțiilor oferite ca suport prin SGBD-ul respectiv.
5. Gestionarea problemelor legate de insonsistența datelor furnizate spre stocare în baza de date prin intermediul unui sistem de excepții specifice

Elementele esențiale cu care lucrează programatorul, furnizate din biblioteca *java.sql* şi bibliotecile corespunzătoare driver-ului furnizat de producătorul bazei de date, sunt:

- *DriverManager* – „centrul de comandă" care înregistrează/activează driverele cu care se lucrează, administrează conexiunile şi tranzacțiile pe partea aplicației;
- *Connection* – elementul ce semnifică conexiunea stabilită cu serverul BD, prin intermediul driver-ului înregistrat în prealabil de către *DriverManager*. Una din responsabilitățile esențiale ale acestui element este constituirea şi execuția comenzilor SQL (obiectelor *Statement*). Într-o aplicație multistrat distribuită,

modul de gestionare a conexiunilor se poate dovedi un factor de scalabilitate deosebit de important.

- *ResultSet* – colecția de înregistrări (rezultatul) regăsite din baza de date. Tot acest element poate gestiona şi pseudo-transparența actualizărilor în baza de date: adică în locul constituirii şi executării unor comenzi (statement) SQL explicite, simpla modificare a valorii unei coloane dintr-o înregistrare a resultset-ului poate determina consistența frazei de actualizare ce va fi transmisă serverului BD fără a nominaliza explicit decât momentul în care să fie executată;
- *Statement* – comanda/fraza SQL ce va fi remisă spre execuție serverului. Pentru a constitui o frază parametrizată este utilizată subclasa *PreparedStatement*, iar corespunzător invocării unei proceduri stocate este folosită subclasa *CallableStatement*.

O imagine sintetică a modului în care comunică aceste elemente este prezentată în figura următoare.

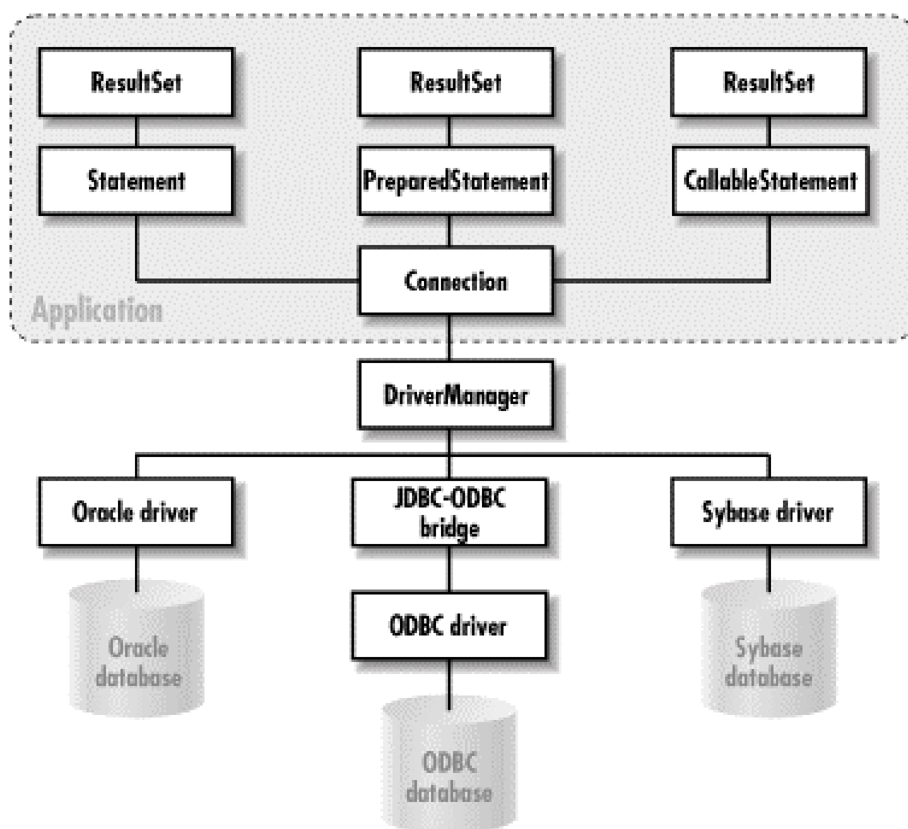



Figura - Elementele care lucrează în cadrul unei sesiuni JDBC, după Flanagan, David Farley, Jim şi Crawford William *Java Enterprise in a Nutshell, 2nd Edition*, O'Reilly, aprilie 2002

Printre puctele forte ale interfeței JDBC trebuie să recunoaştem:

- Existența unui prim nivel de transparență între aplicație şi suportul de stocare relațional datorită eliminării, din perspectiva programatorului aplicației, a sarcinii codificării dialogului cu SGBD-ul printr-un protocol de comunicare proprietar.

- Existența premiselor de portabilitate ale aplicațiilor față de diferitele implementări ale bazelor de date relaționale, având în vedere aici transparența față de structurile de date (diferitele forme de acces şi interogare), mecanismele de securitate şi mecanismele privitoare la tranzacții şi concurență.
- Preocupările pentru îmbunătățirea accesului şi performanțelor prin administrarea resurselor constituite din conexiuni, posibilitățile de transmitere în mod batch a mai multor operațiuni de actualizare acumulate pe parcursul unei sesiuni.
- Gradul larg de acceptabilitate în industrie ceea ce a ridicat JDBC la nivelul de standard veritabil.

Printre problemele încă nerezolvate se pot remarca:

- Nedepăşirea pragului critic al incompatibilității logice dintre modelul aplicațiilor obiectuale şi modelul relațional al structurilor de stocare. JDBC a uniformizat/unificat conceptele legate de organizarea şi accesul structurilor relaționale pe care însă le-a propulsat „ortogonal" în mediile obiectuale Java. Ca urmare, programatorii de aplicații au în față alternative cum ar fi : (1) folosirea brută a noilor concepte/elemente standardizate în JDBC, valorificând avantajele relative ale transparenței față de SGBD-urile proprietare dar păstrând o strânsă dependență față de structurile efective de stocare şi (2) construirea unui strat intermediar de mapare între modelul obiectual al aplicației şi structurile relaționale de stocare aşa încât aplicația să devină transparentă față de schema relațională de implementare a bazei de date. Eventualele modificări ulterioare ale acestor structuri urmează să afecteze doar acest strat. Trebuie menționat că a doua alternativă aduce cu sine o problemă destul de importantă: în cazul aplicațiilor complexe des afectate de modificări în structura bazei de date, activitatea de întreținere a stratului intermediar de mapare poate deveni destul de costisitoare.
- Diferitele implementări proprietare ale bazelor de date relaționale au determinat producătorii de drivere JDBC să includă o serie de extensii specifice care pot aduce îmbunătățiri aplicațiilor bazate pe respectivele platforme de stocare, însă dezideratul portabilității este în mod cert afectat.
- Bazele de date relaționale au evoluat din ce în ce mai mult către un model relațional-obiectual, însă în privința conceptelor promovate de acest model în JDBC nu există o platformă de uniformizare/unificare a acestora. Prin urmare valorificarea noilor structuri de organizare a datelor depinde în mod exclusiv de extensiile proprietare ale standardului prin diferitele drivere furnizate de producători. În acest fel pentru a crea o structură de persistență bazată pe modelul relațional/obiectual trebuie sacrificat în mare parte dezideratul legat de transparența față de SGBD-ul proprietar.

### 2.3.2.2. *Abstractizarea serviciului de persistență: tehnica* **obiectelor de acces la date –** ***DAO***

Responsabilitatea modelului corespunzător stratului pentru persistență constă în asigurarea structurilor de mapare între modelul aplicației şi modelul relațional al bazei de date şi definirea modului de acces.

În acest sens tehnica **DAO** – data access object - se bazează pe delegarea sarcinii reconstituirii obiectelor din structurile relaționale şi salvării eventualelor modificări operate în starea acestora unor clase specifice care stau „în spatele" claselor din domeniul afacerii (*business entity)*. Aceste clase DAO vor accesa apoi interfața JDBC pentru a transforma în fraze SQL operațiile efectuate asupra obiectelor persistente.

Modelul care reflectă modul în care o clasă DAO va asigura accesul în baza de date relațională prin intermediul unui API gen JDBC este prezentat în figura următoare.

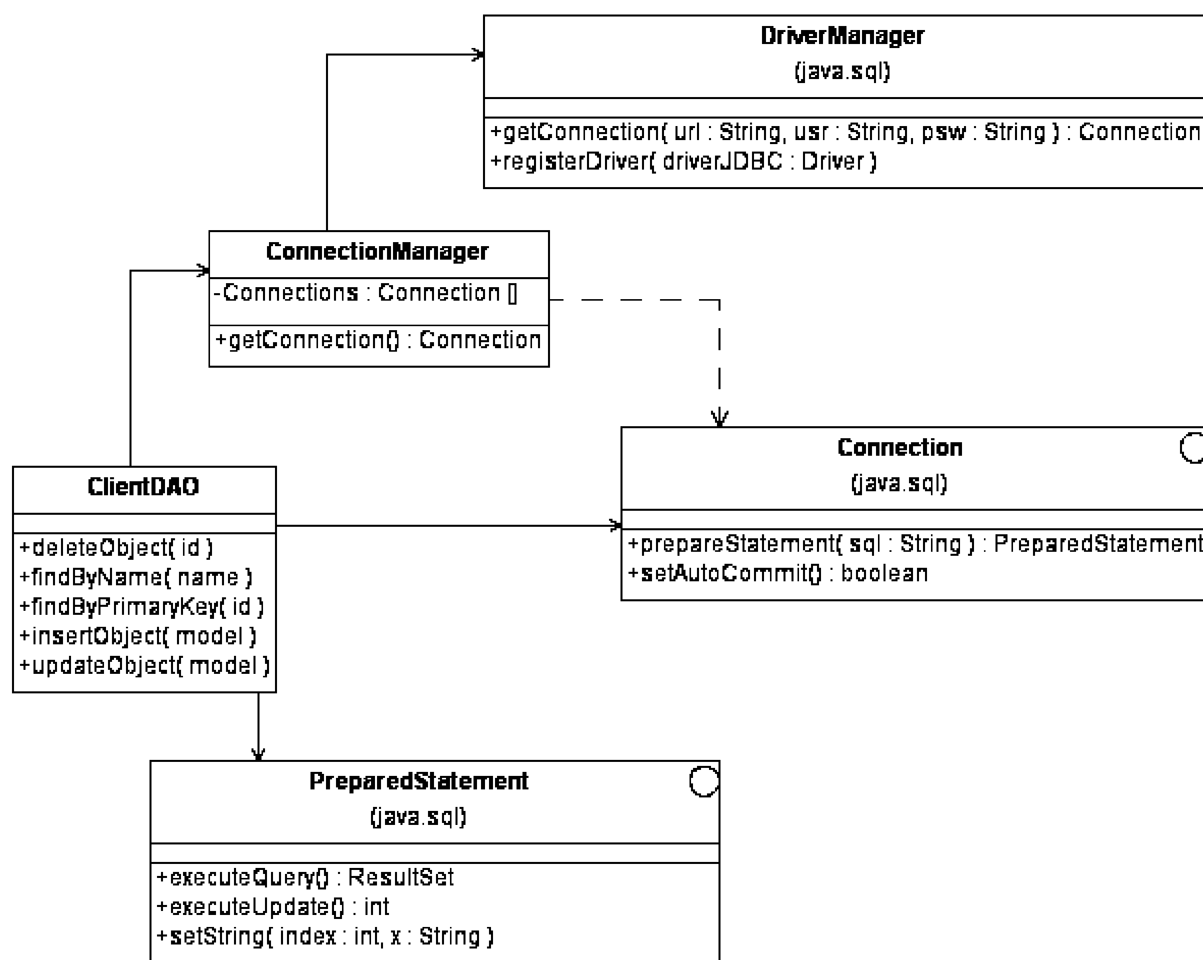



Figura 2-2 Modelul de acces folosind interfețele API-ului JDBC

*ConnectionManager*-ul reprezintă o clasă proprie care gestionează resursele de forma conexiunilor JDBC la serverul bazei de date. Această clasă va invoca clasa *DriverManager* din biblioteca JDBC pentru a obține conexiunile respective pe care le va pune la dispoziția clasei DAO. De asemenea clasa DAO va folosi apoi conexiunile primite de la *ConnectionManager*

pentru a construi obiecte de tip *PreparedStatement* care vor prelua şi executa frazele SQL-DML necesare actualizării datelor din structurile relaționale.

Modul în care sunt utilizată interfețele şi clasele bibliotecii JDBC este ilustrat prin *diagrama de colaborare (de interacțiuni)* din figura următoare.

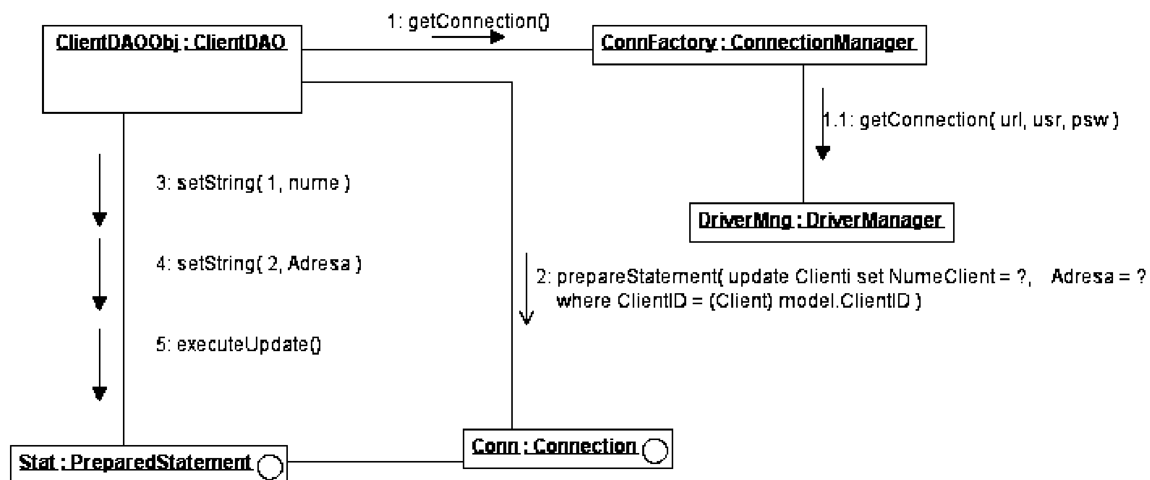
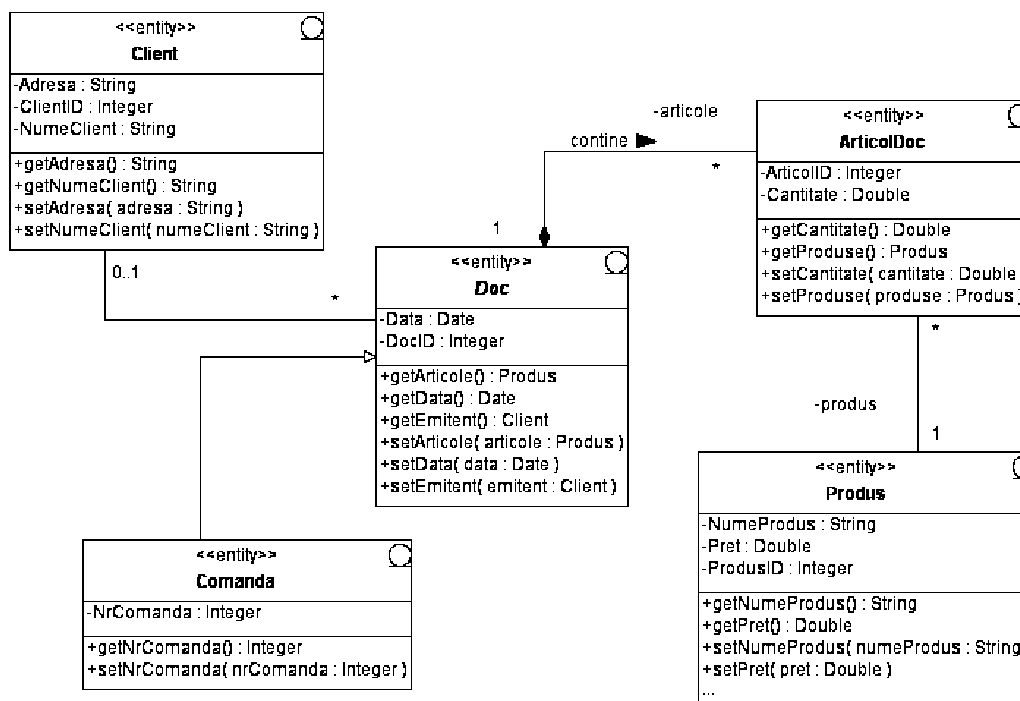



Figura 2-3 Invocarea mecanismelor JDBC pentru salvarea unui obiect Client

***Studiu de caz***

Pentru a exemplifica modalitatea de utilizare a tehnicii DAO într-o aplicație Java, să luăm în considerare un *business model* cum este cel din figura următoare:




Modelul luat în considerare are o structură relativ simplă, care poate fi particularizată prin câteva elemente: asociații, o clasă-asociație, o agregare, o generalizare. Clasele implicate reflectă: *clienții* care pot emite mai multe tipuri de documente, dintre care prin *comenzi* solicită să le fie livrate în anumite cantități diferite *produse*.

Pe baza modelului inițial, pot fi construite tabelele relaționale care să reflecte structurile persistente. Probleme mai deosebite apar în transpunerea relației de generalizare dintre clasele *Doc* şi *Comenzi*. Tehnica urmată în acest sens reflectă aplatizarea verticală a ierarhiei de generalizare, care presupune crearea câte unei tabele specifice atât pentru rădăcina ierarhiei şi pentru fiecare subclasă în parte. De asemenea, în superclasă vor fi mapate drept câmpuri toate atributele comune, iar în subclase va fi păstrată drept cheie primară aceeaşi cheie ca şi în tabela superclasei, care va fi în acelaşi timp şi cheie străină (vezi principiile de mapare discutate în secțiunea anterioară).

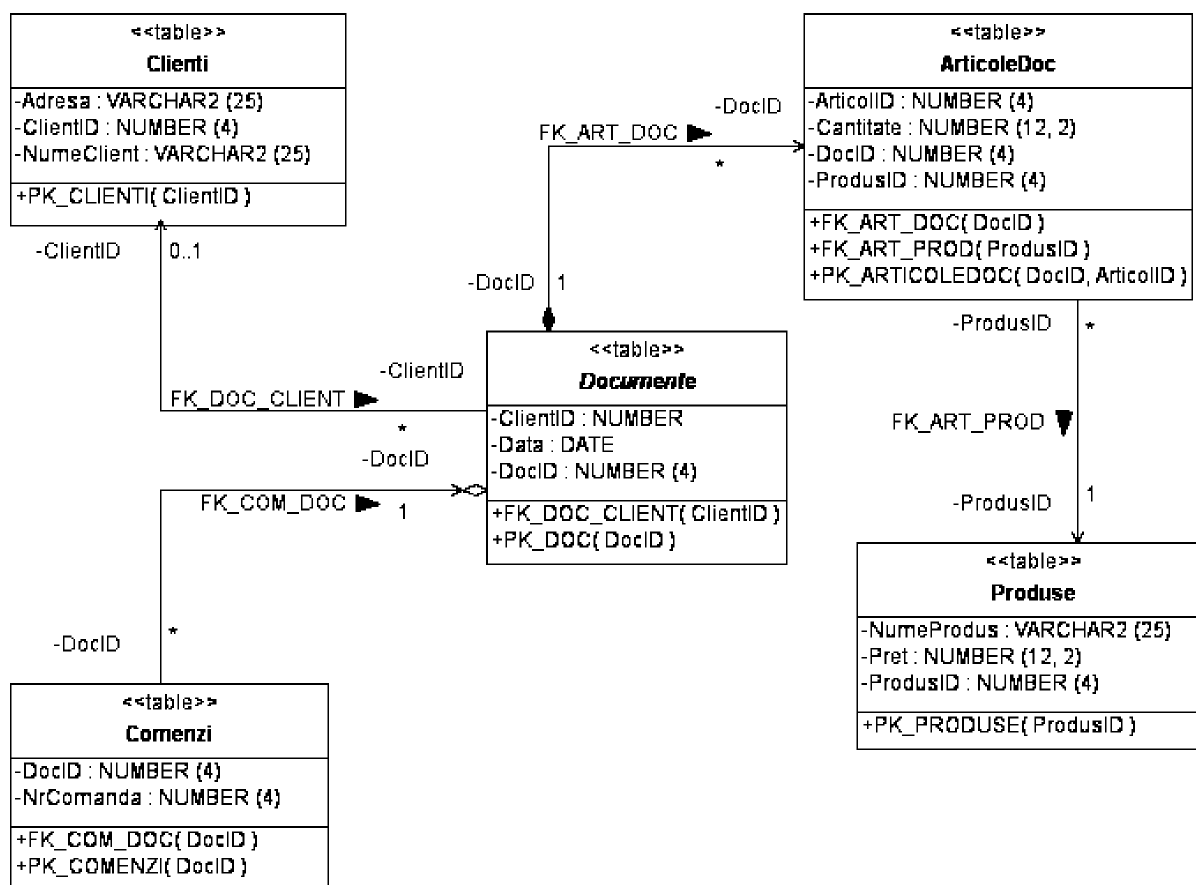



Diagrama de clase corespunzătoare modelului de mapare pentru asigurarea persistenței ar putea fi următoarea.

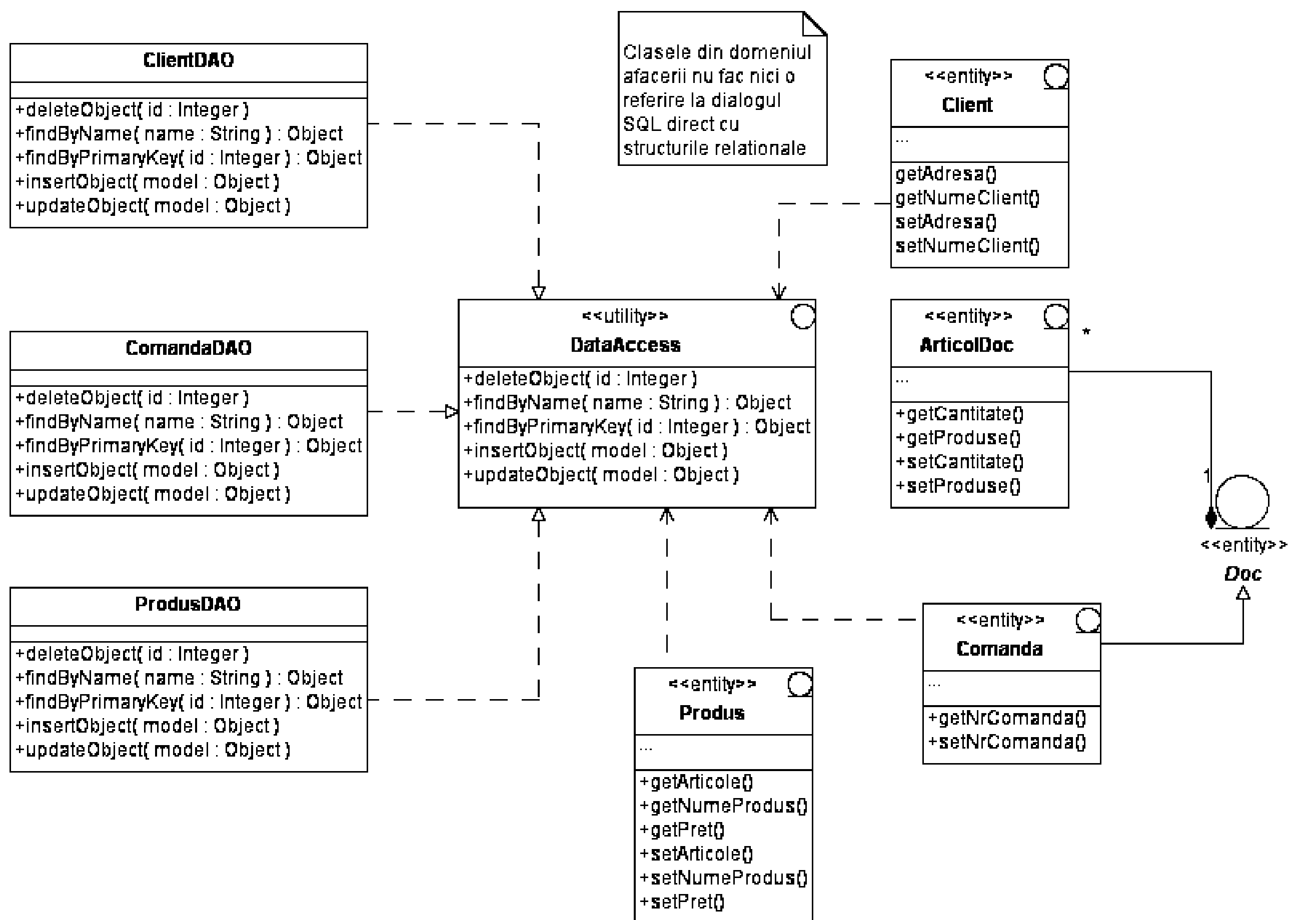



Figura - Modelul DAO pentru asigurarea persistenței

Se poate remarca cu uşurinţă faptul că entităţile din domeniul aplicaţiei lucrează indirect prin intermediul interfeţei DataAccess cu clasele DAO.

Clasele DAO vor asigura astfel stratul care va separa clasele *business entity* de mecanismul efectiv de acces al structurilor relaţionale ale bazei de date.

De remarcat că pentru obiectele *ArticolDoc* nu a fost creată o clasă DAO specifică, sarcinile legate de persistenţă fiind delegate clasei *Comenzi*, care, în virtutea moştenirii clasei *Doc*, se găseşte într-o relaţie de asociere-compunere cu clasa *ArticolDoc*.

Structura internă a clasei ClientDAO este prezentată în listingul următor:

Listing 1: Codificarea JDBC într-o clasă DAO

```
public class ClientDAO implements DataAccess
{
    public void deleteObject( Integer id ) throws DAOAppException, DAOSysException
    {
        Connection conn = ConnectionManager.getConnection();
        try {
            String sqlText = "delete from Clienti where ClientID = ?";
            PreparedStatement dmlStat;
            dmlStat = conn.prepareStatement(sqlText);
            dmlStat.setInt(1, id.intValue());
            dmlStat.executeUpdate();
            dmlStat.close();
        } catch (SQLException e) {}
    }
    public Object findByName( String name ) throws DAOSysException, DAOFinderException
    {
```

```
            Connection conn = ConnectionManager.getConnection();
            try {
                PreparedStatement retriveStat;  // Obiect care va initia dialogul cu BD
                ResultSet rsetClient; // Obiect care va prelua rezultatul dialogului cu BD
                String sqlText = "Select ClientID, NumeClient, Adresa from clienti where NumeClient = ?";
                // Initirea frazei SQL ce va trimisa serverului BD prin intermediul conexiunii stabilite anterior
                retriveStat = conn.prepareStatement(sqlText,
                        ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
                retriveStat.setString(1, name);
                // Initierea dialogului efectiv cu BD si obtinerea cursorului rezultat
                rsetClient = retriveStat.executeQuery();
                rsetClient.absolute(1);
                // Initierea instantei clientului persistent
                Client client = new Client();
                // Preluarea valorilor clientului persistent instantiat in mediul aplicatiei Java
                // din rezultatul dialogului cu baza de date
                client.setClientID(new Integer(rsetClient.getInt(1)));
                client.setNumeClient(rsetClient.getString(2));
                client.setAdresa(rsetClient.getString(3));
                retriveStat.close();
                rsetClient.close();
                // Returnez instanta completa a clientului persistent
                return client;
            } catch (SQLException e) {return null;}
        }
        public Object findByPrimaryKey( Integer id ) throws DAOSysException, DAOFinderException
        {
            Connection conn = ConnectionManager.getConnection();
            try {
                PreparedStatement retriveStat;  // Obiect care va initia dialogul cu BD
                ResultSet rsetClient; // Obiect care va prelua rezultatul dialogului cu BD
                String sqlText = "Select ClientID, NumeClient, Adresa from clienti where ClientID = ?";
                // Initirea frazei SQL ce va trimisa serverului BD prin intermediul conexiunii stabilite anterior
                retriveStat = conn.prepareStatement(sqlText,
                        ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
                retriveStat.setInt(1, id.intValue());
                // Initierea dialogului efectiv cu BD si obtinerea cursorului rezultat
                rsetClient = retriveStat.executeQuery();
                rsetClient.absolute(1);
                // Initierea instantei clientului persistent
                Client client = new Client();
                // Preluarea valorilor clientului persistent instantiat in mediul aplicatiei Java
                // din rezultatul dialogului cu baza de date
                client.setClientID(new Integer(rsetClient.getInt(1)));
                client.setNumeClient(rsetClient.getString(2));
                client.setAdresa(rsetClient.getString(3));
                retriveStat.close();
                rsetClient.close();
                // Returnez instanta completa a clientului persistent
                return client;
            } catch (SQLException e) {return null;}

        }
        public void insertObject( Object model ) throws DAOAppException, DAOUpdateException,
DAOSysException
        {
          Connection conn = ConnectionManager.getConnection();
          try {
            String sqlText =
                "insert into Clienti (clientid, numeclient, adresa) values (?, ?, ?)";
            PreparedStatement dmlStat;
            dmlStat = conn.prepareStatement(sqlText);
            dmlStat.setInt(1, ((Client)model).getClientID().intValue());
            dmlStat.setString(2, ((Client)model).getNumeClient());
            dmlStat.setString(3, ((Client)model).getAdresa());
            dmlStat.executeUpdate();
            dmlStat.close();
            } catch (Exception e) {e.printStackTrace();}
        }
```

```
        public  void  updateObject(Object  model)  throws  DAOAppException,  DAOUpdateException,
DAOSysException
        {
          try {
             Connection conn = ConnectionManager.getConnection();
             String sqlText =
                "update Clienti set numeclient = ?, adresa = ?, codfiscal =? where clientid = ?";
             PreparedStatement dmlStat;
             dmlStat = conn.prepareStatement(sqlText);
             dmlStat.setString(1, ((Client)model).getNumeClient());
             dmlStat.setString(2, ((Client)model).getAdresa());
             dmlStat.setInt(3, ((Client)model).getClientID().intValue());
             dmlStat.executeUpdate();
             dmlStat.close();
             } catch (Exception e) {e.printStackTrace();}
        }
        public Object[] findAll(String sqlWhereClause) throws DAOSysException, DAOFinderException{
             Connection conn = ConnectionManager.getConnection();
             try {
                PreparedStatement retriveStat;
                ResultSet rsetClient;
                Client[] clienti;
                String sqlText = "Select ClientID, numeClient, adresa from clienti";
                if (sqlWhereClause != null) {
                   sqlText += sqlWhereClause;
                }
                retriveStat = conn.prepareStatement(sqlText,
                      ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
                rsetClient = retriveStat.executeQuery();
                rsetClient.last();
                clienti = new Client[rsetClient.getRow()];
                rsetClient.beforeFirst();
                while (rsetClient.next()){
                   clienti[rsetClient.getRow()-1] = new Client();
                   clienti[rsetClient.getRow()-1].setClientID(new Integer(rsetClient.getInt(1)));
                   clienti[rsetClient.getRow()-1].setNumeClient(rsetClient.getString(2));
                   clienti[rsetClient.getRow()-1].setAdresa(rsetClient.getString(3));
                }
                retriveStat.close();
                rsetClient.close();
                // Returnez instanta lista clientilor gasiti
                return clienti;
             } catch (SQLException e) {return null;}
          }
    }
```

Pentru a clarifica cât de cât meritele structurării dialogului cu suportul de stocare (baza de date) în felul în care a fost prezentat prin listingul anterior, trebuie făcute (cel puţin) următoarele observaţii:

- Clasa *ClientDAO* poate fi apreciată ca fiind un producător (*factory*) de instanţe persistente într-o bază de date relaţională.
- Pentru a asigura o anumită independenţă între suportul pentru persistenţă şi logica afacerii, întregul eşafodaj al dialogului cu baza de date este construit în structura clasei DAO şi nu direct în clasa *business entity* corespunzătoare. Astfel pentru a reconstitui în spaţiul tranzient al aplicaţiei un obiect „stocat" în baza de date, este utilizată metoda *findByPrimaryKey()* iar în cazul în care se doreşte obţinerea unei colecţii reprezentând clienţii din baza de date, atunci este utilă metoda *findAll()*.

Clasa *Client* este construită în mare parte respectând convenţiile pentru componentele JavaBeans (atributele sunt private, iar cele care vor constitui interfaţa publică sunt accesibile prin metode *get/set*), metodele *get* sunt folosite pentru a regăsi valorile ce trebuie trimise în baza de

date, iar metodele *set* sunt folosite pentru a stabili valorile obiectelor reconstituite pe baza datelor recuperate din baza de date.

Clasa *ComandaDAO* comportă aspecte mai deosebite datorită faptului că pe baza ei se vor reconstitui şi instanţele clasei *Doc* (toate comenzile reconstituite datorită ierarhiei de generalizare *sunt şi instanţe ale clasei Doc*) şi ale clasei *Articol* (ca urmare a relaţiei de compunere, articolele unui document nu pot exista în afara acestuia, prin urmare crearea, reconstituirea şi ştergerea lor depind întru totul de clasa compozit, în cazul nostru o subclasă – *Comenzi* – a clasei *Doc*).

Listing - Codificarea JDBC într-o clasă DAO

```
public class ComandaDAO implements DataAccess
{
    public void deleteObject( Integer id ) throws DAOAppException, DAOSysException
    {
        Connection conn = ConnectionManager.getConnection();
        try {
          // ComandaDAO are in sarcina si persistenta obiectelor ArticoDoc
          String sqlDmlArticolDoc = "delete from ArticoleDoc where DocID = ?";
          String sqlDmlComanda = "delete from Comenzi where DocID = ?";
          // Structura de generalizare obliga stergerea si din tabela radacinii ierarhice
          String sqlDmlDoc = "delete from Documente where DocID = ?";
          PreparedStatement dmlStat;
          dmlStat = conn.prepareStatement(sqlDmlArticolDoc);
          dmlStat.setInt(1, id.intValue());
          dmlStat.executeUpdate(); //dmlStat.close();
          // mai intai sterg din Documente datorita restrictiei referentiale
          // care leaga ArticoleDoc de Documente
          dmlStat = conn.prepareStatement(sqlDmlDoc);
          dmlStat.setInt(1, id.intValue());
          dmlStat.executeUpdate(); //dmlStat.close();
          dmlStat = conn.prepareStatement(sqlDmlComanda);
          dmlStat.setInt(1, id.intValue());
          dmlStat.executeUpdate();

          dmlStat.close();
        } catch (SQLException e) {}
    }

    public Object findByName( String name ) throws DAOSysException, DAOFinderException
    {
        Integer nr = Integer.valueOf(name);
        Connection conn = ConnectionManager.getConnection();
        try {
          PreparedStatement retriveStat;  // Obiect care va initia dialogul cu BD
          ResultSet rset; // Obiect care va prelua rezultatul dialogului cu BD
          String sqlRetrieveComandaID =
             "Select DocID Comenzi where NrComanda = ?";
          retriveStat = conn.prepareStatement(sqlRetrieveComandaID,
                ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
          retriveStat.setInt(1, nr.intValue());
          // Initierea dialogului efectiv cu BD si obtinerea cursorului rezultat
          rset = retriveStat.executeQuery();
          rset.absolute(1);

          Comanda comanda = (Comanda)findByPrimaryKey(new Integer(rset.getInt(1)));
          return comanda;
        } catch (SQLException e) {return null;}
    }
```

```
    public Object findByPrimaryKey( Integer id ) throws DAOSysException, DAOFinderException
  {
      Connection conn = ConnectionManager.getConnection();
      try {
         PreparedStatement retriveStat;  // Obiect care va initia dialogul cu BD
         ResultSet rset; // Obiect care va prelua rezultatul dialogului cu BD
         String sqlRetrieveDocComenzi =
            "Select D.DocID, C.NrComanda, D.Data, D.ClientID "
            + "from Documente D, Comenzi C where D.DocID = C.DocID and C.DocID = ?";
         String sqlRetrieveArticole =
            "Select ArticolID, ProdusID, Cantitate "
            + "from ArticoleDoc where DocID = ?";
         // Initirea frazei SQL ce va trimisa serverului BD prin intermediul conexiunii stabilite anterior
         retriveStat = conn.prepareStatement(sqlRetrieveDocComenzi,
               ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
         retriveStat.setInt(1, id.intValue());
         // Initierea dialogului efectiv cu BD si obtinerea cursorului rezultat
         rset = retriveStat.executeQuery();
         rset.absolute(1);
         // Initierea instantei clientului persistent
         Comanda comanda = new Comanda();
         // Preluarea valorilor clientului persistent instantiat in mediul aplicatiei Java
         // din rezultatul dialogului cu baza de date
         comanda.setDocID(new Integer(rset.getInt(1)));
         comanda.setNrComanda(new Integer(rset.getInt(2)));
         comanda.setData(Date.valueOf(rset.getString(3)));
         // Reconstitui obiectul (Client) referentiat de atributul Doc.Emitent
         ClientDAO clientAcces = new ClientDAO();
         // Fac apel la clasa DAO corespunzatoare clasei Client
         Client client = (Client)clientAcces.findByPrimaryKey(new Integer(rset.getInt(4)));
         comanda.setEmitent(client);
         // Reconstitui obiectele (ArticolDoc) referentiate de atributul Doc.ArticolDoc
         retriveStat = conn.prepareStatement(sqlRetrieveArticole,
               ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
         retriveStat.setInt(1, id.intValue());
         rset = retriveStat.executeQuery();
         rset.last();
         ArticolDoc[] articole = new ArticolDoc[rset.getRow()];
         Produs produs;
         ProdusDAO produsAcces = new ProdusDAO();
         rset.beforeFirst();
         while (rset.next()){
            articole[rset.getRow()-1] = new ArticolDoc();
            articole[rset.getRow()-1].setArticolID(new Integer(rset.getInt(1)));
            articole[rset.getRow()-1].setCantitate(new Double(rset.getDouble(3)));
            // Fac apel la clasa DAO corespunzatoare clasei Produs
            produs = (Produs)produsAcces.findByPrimaryKey(new Integer(rset.getInt(2)));
            articole[rset.getRow()-1].setProdus(produs);
         }
         retriveStat.close(); rset.close();
         // Returnez instanta completa a Comenzii persistente
         return comanda;
      } catch (SQLException e) {return null;}

  }

public void insertObject(Object model) throws DAOAppException, DAOUpdateException,
      DAOSysException
  {
    Connection conn = ConnectionManager.getConnection();
    try {
       String sqlDmlArticolDoc =
```

```
            "insert into ArticoleDoc (ArticolID, DocID, ProdusID, Cantitate) values (?, ?, ?, ?)";
        String sqlDmlComanda =
            "insert into Comenzi (DocID, NrComanda) values (?, ?)";
        String sqlDmlDoc =
            "insert into Documente (DocID, Data, ClientID) values (?, ?, ?)";
        PreparedStatement dmlStat;
        // Mai intai inregistrarea in tabela de generalizare (corespunsatoare clasei parinte)
        dmlStat = conn.prepareStatement(sqlDmlDoc);
        // Upcastez la nivelul clasei Doc, datorita generalizarii
        // Comanda este un Doc, iar parametrul model implicit este si Doc si Comanda
        dmlStat.setInt(1, ((Doc)model).getDocID().intValue());
        dmlStat.setDate(2, ((Doc)model).getData());
        // Parcurg lantul de referinte Doc.Emitent -> Client.ClientID
        dmlStat.setInt(3, ((Doc)model).getEmitent().getClientID().intValue());
        dmlStat.executeUpdate(); //dmlStat.close();
        // Apoi inserare si in Comenzi
        dmlStat = conn.prepareStatement(sqlDmlComanda);
        dmlStat.setInt(1, ((Doc)model).getDocID().intValue()); // ID-ul este preluat de la nivelul clasei
parinte
        dmlStat.setInt(2, ((Comanda)model).getNrComanda().intValue());
        dmlStat.executeUpdate(); //dmlStat.close();
        // In cele din urma tb. rezolvate si Articolele
        dmlStat = conn.prepareStatement(sqlDmlArticolDoc);
        // Trebuie parcurse toate articolele ale caror referinte pot final obtinute
        // din atributul tip array al clasei Doc
        for (int i = 0; i<((Doc)model).getArticole().length; i++){
            ArticolDoc[] articole = ((Doc)model).getArticole();
            dmlStat.setInt(1, articole[i].getArticolID().intValue());
            dmlStat.setInt(2, ((Doc)model).getDocID().intValue());
            dmlStat.setInt(3, articole[i].getProdus().getProdusID().intValue());
            dmlStat.setDouble(4, articole[i].getCantitate().doubleValue());
            dmlStat.executeUpdate();
        }
        dmlStat.close();
        } catch (Exception e) {e.printStackTrace();}
    }

    public void updateObject( Object model ) throws DAOAppException, DAOUpdateException,
       DAOSysException
    {
      Connection conn = ConnectionManager.getConnection();
      try {
        String sqlDmlArticolDoc =
            "update ArticoleDoc set DocID=?, ProdusID=?, Cantitate=? where ArticolID = ?";
        String sqlDmlComanda =
            "update Comenzi set NrComanda = ? where DocID=?";
        String sqlDmlDoc =
            "update Documente set Data=?, ClientID=? where DocID=?";
        PreparedStatement dmlStat;
        // DOCUMENTE
        dmlStat = conn.prepareStatement(sqlDmlDoc);
        dmlStat.setInt(3, ((Doc)model).getDocID().intValue());
        dmlStat.setDate(1, ((Doc)model).getData());
        dmlStat.setInt(2, ((Doc)model).getEmitent().getClientID().intValue());
        dmlStat.executeUpdate(); //dmlStat.close();
        // COMENZI
        dmlStat = conn.prepareStatement(sqlDmlComanda);
        dmlStat.setInt(2, ((Doc)model).getDocID().intValue());
        dmlStat.setInt(1, ((Comanda)model).getNrComanda().intValue());
        dmlStat.executeUpdate(); //dmlStat.close();
        // ARTICOLEDOC
        dmlStat = conn.prepareStatement(sqlDmlArticolDoc);
        for (int i = 0; i<((Doc)model).getArticole().length; i++){
```

```
                ArticolDoc[] articole = ((Doc)model).getArticole();
                dmlStat.setInt(4, articole[i].getArticolID().intValue());
                dmlStat.setInt(1, ((Doc)model).getDocID().intValue());
                dmlStat.setInt(2, articole[i].getProdus().getProdusID().intValue());
                dmlStat.setDouble(3, articole[i].getCantitate().doubleValue());
                dmlStat.executeUpdate();
            }
            dmlStat.close();
            } catch (Exception e) {e.printStackTrace();}
        }
    }
```

## *2.4. Expunerea domeniului afacerii: interfața grafică. Framework-ul Java Swing*

Interfețele grafice moderne se bazează pe o serie de componente specifice pornind de la ferestre (window) ce conțin o diversitatea de aşa-numite "controale grafice", în fapt componente care au un aspect mai mult sau mai puțin profesionist funcție de calitatea claselor din bibliotecile fundamentale din care provin. O altă caracteristică esențială a interfețelor grafice actuale este că nu sunt construite într-o paradigmă algoritmică prin care să fie structurată prelucrarea unor *stream*-uri care provin direct de la tastatura utilizatorului, sau trimit anumite şiruri de caractere la ecranul consolei, ci se bazează pe un nou "stil" de programare - *programarea bazată pe evenimente* adoptată de toate mediile de dezvoltare vizuală.

Într-un *sistem bazat pe evenimente*, aplicația "aşteaptă să se întâmple ceva" în mediul de execuție, cu alte cuvinte aşteaptă producerea unui *eveniment*. Când apare un astfel de eveniment, aplicația răspunde respectivului eveniment după care îl aşteaptă pe următorul. Pe scurt, aceasta este esența *modelului bazat evenimente*.

În JAVA (în special de la JDK 1.3) clasele care sunt folosite pentru construirea interfețelor grafice utilizator sunt furnizate prin intermediul unei biblioteci API ce conține în principal componente vizuale numite *Swing* şi care extind cu noi capabilități mai vechea bibliotecă (care există bineînțeles şi în distribuțiile actuale) *AWT*.

Obiectele din package-urile bibliotecii *javax.swing* sau *java.awt* sunt numite obiecte GUI (graphical user interface) şi sunt valorificate, după cum am amintit mai sus, într-un stil de programare *bazat pe evenimente*, evenimente care însoțesc (inter)acțiunile utilizatorilor cu obiectele grafice.

### 2.4.2. Componentele grafice – bibliotecile java.awt şi javax.swing

O aplicație cu o interfață bazată pe ferestre (window) se manifestă sub forma unui "panou" conținând un grup de opțiuni grafice simbolizate prin meniuri, butoane, căsuțe textuale etc. După afişarea unei astfel de interfețe aplicația va aştepta o interacțiune din partea utilizatorului care poate "apăsa" un buton, poate alege o opțiune dintr-un meniu sau poate introduce text într-o căsuță de editare. Atunci când utilizartorul realizează respectiva acțiune aplicația va răsupunde evenimentului produs după care reintră în starea de aşteptare.

Sistemele bazate pe ferestre (cum sunt sistemele MSWindows sau Motif®) pot detecta "obiecte eveniment" cum sunt click-urile de mouse, mişcarea cursorului mouse-ului, apăsarea tastelor şi gestionează afişarea interfeţelor grafice. O aplicaţie JAVA interacţionează cu sistemul window nativ prin intermediul componentelor *AWT* - Abstract Window Toolkit.

#### 2.4.2.1. Scurtă descriere a modului de dezvoltare a unei interfeţe grafice simple cu cadre, butoane şi evenimente, bazată pe biblioteca java.awt

Ferestrele din cadrul aplicaţiilor GUI în Java sunt de două tipuri: cadre -frames- şi dialoguri:

- Un *frame* este o fereastră cu scop general prin intermediul cărei utilizatorii interacţionează în mod obişnuit cu aplicaţia. Într-o aplicaţie există cel puţin un *frame* care va servi drept fereastra principală.
- Un *dialog* este o fereastră cu scop limitat destinată în primul rând afişării unor informaţii cum sunt mesajele de eroare sau pentru preluarea unor răspunsuri simple (standard) gen *da* sau *nu*.

Corespunzător acestor tipuri de ferestre există două clase specifice *Frame* şi *Dialog*. Clasa Frame conţine funcţionalităţile rudimentare necesare oricărei ferestre cum ar fi minimizarea, mutarea, redimensionare etc. Dezvoltatorii îşi vor crea propriile ferestre (formulare) ale aplicaţiilor lor pe baza (sau derivând) această clasă fundamentală.

La fel ca şi în cazul clasei *Frame*, clasa *Dialog* conţine funcţionalităţile rudimentare necesare unei ferestre standard de dialog. Această clasă va sta la baza tuturor ferestrelor gen *MessageBox* necesare într-o aplicaţie.

Stabilirea principalelor trăsături pentru ferestrele tip *frame* se realizează pe baza următoarelor metode:

- *setsize(*lăţime, lungime*)* pentru stabilirea dimesiunilor;
- *setResizeable(*true|flase*)* pentru a activa sau dezactiva posibilitatea redimesionării ferestrei de către utilizator;
- *setTitle(*text*)* pentru desemna un titlu afişat în bara superioară a ferestrei;
- *setLocation(x, y)* pentru a desemna originea colţului stânga-sus al ferestrei, de remarcat faptul că originea (0, 0) de la care sunt consideraţi factorii x şi y este colţul stânga sus al cadrului superior şi nu cel din stânga-jos;
- *setVisible(*true|flase*)* pentru a afişa sau ascunde fereastra, echivalent cu metodele *show()* şi *hide(),* recomandate a fi înlocuite cu această metodă.

Menirea unui unui astfel de frame este să găzduiască obiectele GUI cu care va interacţiona utilizatorul aplicaţiei. Distribuirea, localizarea sau poziţionarea obiectelor într-o fereastră poate fi controlată folosind un anumit model de gestionare -*layout manager*- care este desemnat prin metoda *setLayout()*. Dacă se optează pentru poziţionarea absolută (fără nici un factor de relativizare a obiectelor funcţie de mărimea ferestrei de exemplu) atunci metoda *setlayout()* este apelată în felul următor:

```
setLayout(null);
```

În acest fel obiectele (butoane, căsuțe de text, grupuri de opțiuni etc.) vor fi afişate întotdeauna la coordonatele stabilite de programator. Spre exemplu poziționarea şi dimensionarea unui boton de comandă (clasa *java.awt.Button* sau *java.swing.JButton* în Swing) se realizează prin metoda:

```
setBounds(x, y, lățime, lungime);
```

Plasarea unor componente GUI într-un cadru implică în Java două declarații sau specificații:

1. declararea în clasa derivată din *Frame* (sau *JFrame*) a membrilor care desemnează componentele grafice ce vor fi afişate la execuție;
2. la instanțierea ferestrei respective (sau mai exact a clasei derivate din *Frame*) se va specifica explicit (în cadrul constructorului) instanțierea componentelor grafice şi apoi afişarea lor prin metoda *add()*:

```
add(InstanțăButon);
```

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class FrameTest extends Frame implements ActionListener, WindowListener{
// Declar doua componente care vor apartine cadrului FrameTest
  Button b1;
  Button b2;
// In constructorul FrameTest activez butoanele, fereastra coresponzatoare
// cadrului FrameTest, obiectele receptoare pentru evenimetele butoanelor
// si cele receptoare pentru evenimentele ferestrei
  public FrameTest(){
// stabilsc titlul ferestrei
    this.setTitle("Frame Test");
// inregistrez obiectul care va trata evenimentele ferestrei
// adica (insusi cadrul TestFrame)
    this.addWindowListener(this);
    setLayout(null);
// instantiez butoanele, le stabilesc dimensiunile,
// inregistrez obiectele care vor trata evenimentele lor (cadrul FrameTest)
// si le activez in cadrul FrameTest
    b1 = new Button("OK");
    b2 = new Button("Cancel");
    b1.setBounds(50, 100, 100, 50);
    b2.setBounds(50, 300, 100, 50);
    b1.addActionListener(this);
    b2.addActionListener(this);
    add(b1);
    add(b2);
// stabilesc dimensiunile ferestrei cadrului FrameTesti o afisez
    setSize(400, 400);
    setVisible(true);
  }
// metoda in care tratez evenimentele butonului b1 care va inchide fereastra
// si va incheia procesul de executie
  public void actionPerformed(ActionEvent event){
    if (event.getSource().equals(b1))
      setVisible(false);
      System.exit(0);
```

```
  }
// Implementez metodele cerute de interfata WindowListener
// la care se conformeaza obiectul care trateaza evenimentele ferestrei
// (adica insusi cadrul FrameTest)
  public void windowActivated(WindowEvent event){}
  public void windowClosing(WindowEvent event){
    System.exit(0);
  }
  public void windowClosed(WindowEvent event){}
  public void windowDeactivated(WindowEvent event){}
  public void windowIconified(WindowEvent event){}
  public void windowDeiconified(WindowEvent event){}
  public void windowOpened(WindowEvent event){}
// metoda main care instantiaza cadrul TestFrame, afisind astfel fereastra
// cu cele doua butoane
  public static void main(String[] args) {
    FrameTest frm = new FrameTest();
    System.out.println("Asta e");
  }
}
```

Şi rezultatul va fi:

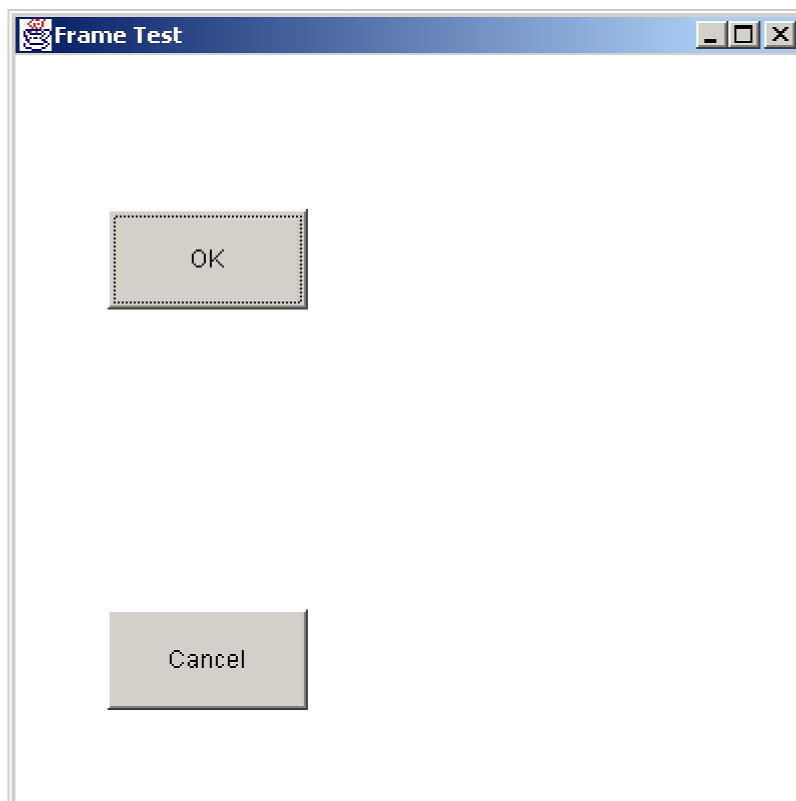



Figura - Fereastra obţinută folosind clasa *java.awt.Frame*

### *2.4.2.2. Componente grafice SWING*

Componentele Java Swing sunt definite într-o bibliotecă de clase conţinută în package-ul javax.swing şi în subpackage-urile acestuia. O aplicaţie poate folosi aceste clase pentru a construi şi gestiona o interfaţa grafică utilizator. Una dintre clasele fundamentale din această

bibliotecă este clasa abstractă *JComponent*, majoritatea componentelor grafice folosite în Java sunt instanțe ale acesteia.

O parte din subclasele elementare derivate din *JComponent* sunt următoarele:

Tabelul 2.1 Componente din biblioteca *javax.swing*

| **Componenta** | **Descriere** |
|---|---|
| ***JButton*** | Un simplu buton de comandă care poate fi apăsat printr-un click de mouse |
| ***JCheckBox*** | Un buton care poate fi "bifat" |
| ***JComboBox*** | O listă drop-down clasică care poate conține opțional şi un camp textual editabil. Utilizatorul poate de asemenea selecta o valoare din listă care va fi afişată in câmpul textual |
| ***JFileChooser*** | O componentă care asigură un mecanism simplu pentru selectarea unui fişier |
| ***JLabel*** | O componentă care conține un şir textual sau o imagine (nu reacționează la evenimente de introducere/editare text) |
| ***JList*** | O componentă care permite utilizatorului să selecteze unul sau mai multe dintre elementele care le conține |
| ***JRadioButton*** | Un buton de stare on/off. De obicei sunt organizate în grupuri dintre care unul singur poate fi selectat pe poziția "on" |
| ***JScrollPanel*** | O componentă care gestionează o porțiune vizualizabilă prin derularea conținutului în cadrul unui view |
| ***JSlider*** | O componentă care permite utilizatorului să selecteze o valoare prin rularea unui indicator pe o scală (asemenea unui "potențiometru") |
| ***JTextArea*** | O zonă care poate găzdui mai multe linii de text editabile sau nu |
| ***JTextField*** | O zonă pentru introducerea unei singure linii de text |

De fapt *JComponent* este o subclasă a *java.awt.Component* şi va moşteni astfel multe dintre cele mai evidente proprietăți ale acestei clase. Proprietățile generice ale unei componente includ colorile pentru *background* şi *foreground*, locația în cadrul gazdă şi dimensiunea. Valorile acestora pot fi obținute prin metodele:

```
public Color getForeground () ;
public Color getBackground () ;
public Point getLocation () ;
public Dimension getSize () ;
public Rectangle getBounds() ;
```

şi pot fi stabilite prin metodele:

```
public void setForeground (Color fg) ;
public void setBackground (Color bg) ;
public void setLocation (Point p) ;
public void setSize (Dimension d) ;
public void setBounds(int x,int y,int Width,int Height) ;
```

*Color, point*, *Dimension* şi *Rectangle* sunt clase AWT (definite în biblioteca *java.awt*). Clasa *Color* deţine un număr de referinţe de constante cum ar fi Color.red, Color.blue, Color.green.

O instanţă a clasei *Point* reprezintă o poziţie relativă într-un spaţiu determinat prin coordonate x-y. Unităţile sunt în pixeli, iar originea (0, 0) înseamnă colţul din dreapta sus. Atributele unui *Point* pot fi accesate prin variabilele (membrii) publici *x* şi *y* de tip **int**. Acestă metodă are şi metode *getX* şi *getY* care returnează **double**, dar nu are metode *setX* şi *setY*.

O instanţă a clasei *Dimension* încapsulează lăţimea şi înălţimea, de asemenea în pixeli. Atributele pot fi accesate la fel ca şi în cazul clasei *Point* prin membrii publici *height* şi *width*. La fel există metodele:

```
public double getHeight()
public double getWidth()
```

O instanţă a clasei *Rectangle* specifică o zonă într-un spaţiu de coordonate, astfel: membrii *height* şi *width* specifică înălţimea respectiv lăţimea, iar membrii *x* respectiv *y* specifică coordonatele obiectului relativ la părintele său (cel care îl conţine).

**Containere**

Un obiect care poate conţine componente se numeşte ***container***. Acestea sunt modelate de către clasa abstractă *java.awt.**Container***. Un aspect important, care simplică lucrul cu intefeţele grafice, constă în faptul că această clasă este, la rândul ei, derivată din clasa *Component*. Prin urmare un *Container* poate conţine un alt *Container*. De asemenea clasa Swing *JComponent* este o subclasă a clasei *Container*. Cum orice *JComponent* poate conţine alte componente putem spune că:

- O **componentă** reprezintă un element distinct al unei interfeţe grafice utilizator, cum ar fi un buton, un câmp textual etc.
- Un **container** reprezintă o componentă a interfeţei grafice utilizator care poate conţine alte componente.

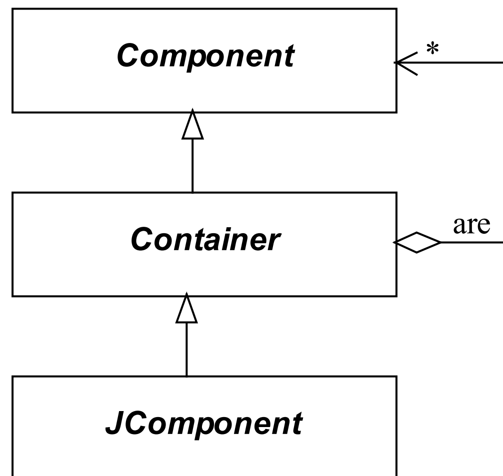



Figura 2-4 Relațiile *Component* → *Container* → *JComponent*

Cel mai simplu şi „curat” container este *JPanel*. Un *JPanel* este folosit în general ca o regiune simplă în care sunt aduse şi grupate o colecție de alte componente. Componentele sunt adăugate unui container prin metoda *add()*. De exemplu setul următor de instrucțiuni creează un *JPanel* şi adaugă două butoane:

```
JPanel p = new JPanel;
p.add(new JButton(”Ok”));
p.add(new JButton(”Cancel”));
```

*Containere top-level*

Un container top-level este un container care nu este inclus în nici un alt container. Clasele Swing *JApplet*, *JDialog*, *JFrame* şi *JWindow* reprezintă containerele top-level din Swing.

Container-ul standard pentru o aplicație de interfață grafică este *JFrame*.Un *JFrame* este o fereastră cu titlu şi margini bine definite care poate fi mutată, redimensionată, minimizată (iconificată) etc. De asemenea un *JFrame* poate avea o bară de meniu. Sunt însă două lucruri mai deosebite ce merită menționate în legătură cu această clasă: deşi este o clasă derivată în primul rând din *java.awt.Container* nu este totuşi o subclasă a *JComponent*, iar în al doilea rând aceasta delegă responsabilitatea gestiunii componentelor sale unui alt obiect de tip *JRootPane*.

Un *JRootPane* derivă din *JComponent* şi are ca principală responsabilitate gestionarea conținutul unui alt container. Un astfel de obiect este un compozit incluzând printre altele un (panou) *content pane* care de obicei este un *JPanel* şi reprezintă aria de lucru dintr-un *JFrame*, excluzând titlul, marginile şi meniul. Lucrurile par destul de complicate însă ceea ce trebuie reținut este faptul că un *JFrame* are un *JRootPane* care conține un „panou de componente” (un „*content pane”*). Pentru a obține referința panoului de componente al unui *JFrame* se apelează la metoda *getContentPane()* care returnează un obiect de tip *java.awt.Container*. Prin urmare pentru a adăuga, de exemplu, un buton pe un *JFrame* vom proceda cam în felul următor:

```
JFrame f = new JFrame(”O fereastra oarecare”);
JButton b = new JButton(”Gata”);
Container cp = f.getContentPane(); // sau JPanel cp = f.getContentPane();
```

```
    Cp.add(b);
```

*JApplet*, *JDialog*, *JWindow* şi *JInternalFrame* de asemenea folosesc un *JRootPane* pentru a-şi gestiona componentele lor. Toate aceste clase implementează interfaţa *RootPaneContainer*.

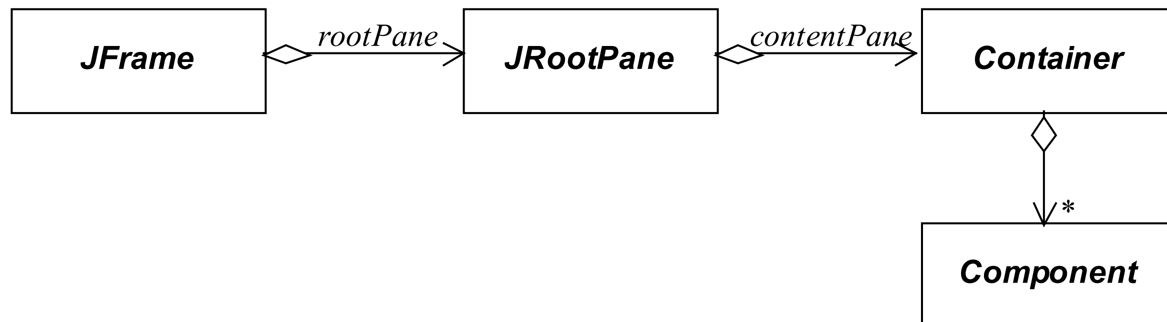



Figura - Modul în care *JFrame* delegă responsabilitatea administrării componentelor unui *JRootPane*

Instanţele claselor *JApplet*, *JDialog*, *JFrame* şi *JWindow* sunt numite componente *grele* (heavy-weight) în opoziţie cu instanţele subclaselor clasei *JComponent* care sunt considerate *uşoare*. Atunci când este creată o componentă „grea" o altă componentă este de asemenea creată în mediul nativ GUI în care rulează aplicaţia Java, componentă numită pereche - *peer*. Componenta pereche este parte a sistemului grafic *window* nativ şi are rolul de a captura de fapt acţiunile utilizatorilor şi de a administra zona afişată pe ecran în care este prezentată componenta Java.

Pe de altă parte componentele „uşoare" sunt implementate complet în Java. Ele nu au asociate obiecte pereche din mediu grafic *window* nativ. Acestea sunt afişate în spaţiul furnizat de către containerele părinţilor lor „grei".

Obiectele „pereche" fiind „în spatele" claselor AWT cu care au legătură nu necesită o aenţie deosebită din partea programatorului Java.

Component
Container
Window
Frame
JComponent
JFrame
<<interface>>
RootPaneContainer
AbstractButton
JComboBox
JLabel
JPanel
JRootPane
JScrollPane
JSlider
javax.swing.text
JTextComponent
JButton
JToggleButton
JCheckBox
JRadioButton
JTextField
JTextArea

Figura 2-5 O parte din clasele componentelor Swing

#### *2.4.2.3. Folosirea componentelor grafice şi programarea bazată pe evenimente*

După cum vom observa în continuare, construirea unui formular (*JFrame*) împreună cu componentele sale (*JComponents)*, precum şi interacțiunile bazate pe evenimente (*Event*) cu aceste componente, au un „aer” mult mai „obiectual” decât în alte medii de programare, ceea ce conduce, în primul rând, la un spor de flexibilitate.

**Obținerea unui simplu JFrame**

Din exemplul prezentat în secțiunea introductivă (referitoare la biblioteca AWT) se pot desprinde câteva mecanisme de programare în ceea ce priveşte componentele care sunt valabile în linii mari şi pentru componentele Swing.

Astfel, componenta *JFrame* este instanțiabilă într-un mod asemănător componenta *Frame* din AWT, din care derivă de fapt, diferența constând în faptul că stabilirea tittlului frame-ului se poate face în momentul instanțierii nefiind nevoie de metoda *setTitle(String)*:

```
JFrame f = new JFrame(”JFrame Test”);
```

De asemenea pentru stabilirea dimensiunilor clasa *JFrame* moşteneşte metoda *setSize(int, int)* de la clasa *Component*, iar pentru a afişa un *JFrame* acesta este făcut vizibil prin metoda *setVisible(boolean)*. Iată cel mai simplu program de afişare a unui *JFrame*:

```
public class DisplayJFrame {
  public static void main(String[] args) {
    JFrame f = new JFrame("JFrame Test");
    f.setSize(300, 200);
```

```
        f.setVisible(true);
    }
}
```

Însă la fel ca şi în caz ul *Frame*-ului părinte, nici *JFrame* nu încheie procesul în care se execută la închiderea ferestrei prin meniul ataşat sau din butonul „close”.

**Adăugarea componentelor într-un *JFrame***

După cum am arătat mai înainte, clasa *JFrame* este un container care îşi delegă responsabilitatea gestiunii componentelor sale unui *JRootPane*. Prin urmare pentru a adăuga un buton acesta trebui plasat de fapt pe „panoul de componente” cam în felul următor:

```
JFrame f = new JFrame("JFrame Test");
f.setSize(300, 200);
f.setVisible(true);
JButton b = new JButton("OK");
Container cp = f.getContentPane();
cp.add(b);
```

Adăugarea unei componente nu este şi condiţia suficientă pentru afişarea acesteia. Modul cum va fi „expusă” şi unde va fi expusă respectiva componentă cade în sarcina unui alt obiect cu care este echipat fiecare container, şi anume *layoutManager*-ul. Prin urmare un container delegă responsabilitatea poziţionării (dispunerii) şi dimensionării componentelor unui obiect de tip *layout maager*, care trebuie să implementeze interfaţa *java.awt.LayoutManager*. Această interfaţă specifică metodele tuturor tipurilor de *layout manager*. În unele cazuri este folosită interfaţa *java.awt.LayoutManager2* care extinde interfaţa originală *LayoutManager*.

Pentru accesarea şi configurarea propriului *layout manager* un container are la dispoziţie metodele *getLayout* şi *setLayout*:

```
public LayoutManager getLayout();
public void setLayout(LayoutManager manager);
```

Distribuţia Java furnizează mai multe clase care implementează interfaţa *LayoutManager* printre care *FlowLayout, BorderLayout, GridLayout, CardLayout, GridBagLayout, BoxLayout, OverlayLayout*. Unele se găsesc în package-ul *javax.swing*, iar altele în *java.awt*. Iată caracteristicile de bază ale layout manager-ilor standard:

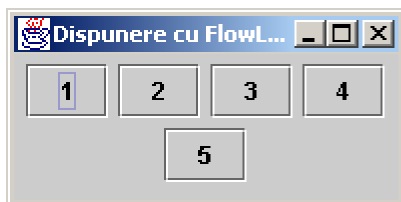
| **Componenta** | **Descriere** |
|---|---|
| ***FlowLayout*** | Dispune sau aşează componentele de la stânga spre dreapta, de sus în jos. Este layout manager-ul implicit(default) pentru *JPanel* |
| ***BorderLayout*** | Afişează până la cinci componente, poziţionate ca “north” – nord, “south” – sud, “east” – est, “west” – vest şi “center” –centru. Este layout manager-ul implicit pentru panoul de componente al *JFrame*. |
| ***GridLayout*** | Aşează componentele într-un grid bidimensional. |
| ***CardLayout*** | Componentele sunt afişate pe rând, dispuse fiind prin suprapunere (parţială). |
| ***GridBagLayout*** | Afişează componentele vertical şi orizontal funcţie de un set re restricţii specifice. Este cel mai complex şi mai flexibil layout manager. |

| | |
|---|---|
| ***BoxLayout*** | Afişează componentele fie o singură linie orizontală fie pe o singură coloană verticală. Este layout-managerul implicit pentru containerul *Box* din biblioteca Swing. |
| ***OverlayLayout*** | Afişează componentele aşa încât referinţele de aliniere ale lor indică acelaşi loc. Prin urmare sunt dispuse sub forma unei stive: unele deasupra celorlalte. |

Iată în continuare câteva exemple privind modul cum sunt afişate componentele folosind diverse tipuri *LayoutManager*:

```
public class DisplayFrameFlowLayout {
  public static void main(String[] args) {
    JFrame f = new JFrame("Dispunere cu FlowLayout");
    JButton b1 = new JButton("1");
    JButton b2 = new JButton("2");
    JButton b3 = new JButton("3");
    JButton b4 = new JButton("4");
    JButton b5 = new JButton("5");
    Container cp = f.getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(b1);
    cp.add(b2);
    cp.add(b3);
    cp.add(b4);
    cp.add(b5);
    f.setSize(200, 100);
    f.setVisible(true);
  }
}
```

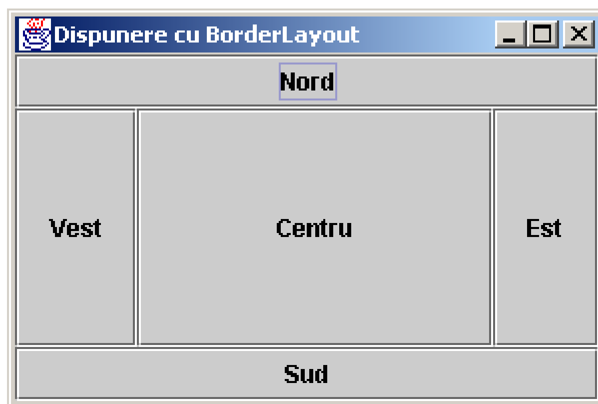
Rezultatul:




```
import javax.swing.*;
import java.awt.*;

public class DisplayFrameBorderLayout {
  public static void main(String[] args) {
    JFrame f = new JFrame("Dispunere cu BorderLayout");
    Container cp = f.getContentPane();
    cp.setLayout(new BorderLayout());
    cp.add(new JButton("Nord"), BorderLayout.NORTH);
    cp.add(new JButton("Sud"), BorderLayout.SOUTH);
    cp.add(new JButton("Est"), BorderLayout.EAST);
    cp.add(new JButton("Vest"), BorderLayout.WEST);
    cp.add(new JButton("Centru"), BorderLayout.CENTER);
    f.setSize(300, 200);
    f.setVisible(true);
  }
}
```

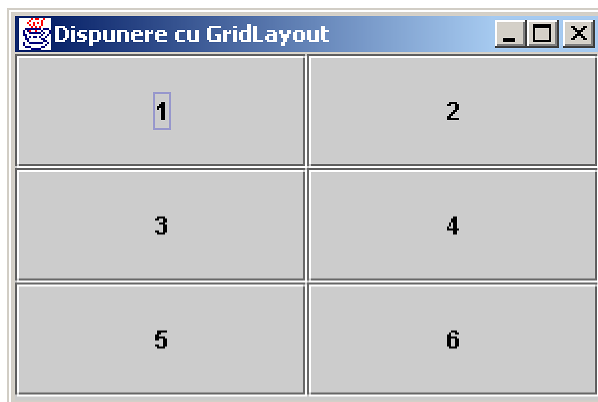
Rezultatul:




```
import javax.swing.*;
import java.awt.*;

public class DisplayFrameGridLayout {
  public static void main(String[] args) {
    JFrame f = new JFrame("Dispunere cu GridLayout");
    Container cp = f.getContentPane();
    cp.setLayout(new GridLayout(3, 2));
    cp.add(new JButton("1"));
    cp.add(new JButton("2"));
    cp.add(new JButton("3"));
    cp.add(new JButton("4"));
    cp.add(new JButton("5"));
    cp.add(new JButton("6"));
    f.setSize(300, 200);
    f.setVisible(true);
  }
}
```

Rezultat:




Modificările asupra grupului de componente dintr-un container după afişarea acestuia, va determina invalidarea lui. Aceast fapt poate fi verificat prin metoda *isValid()* care va returna o valoare booleană. Revalidarea unui container echivalează cu reafişarea lui şi se realizează prin metoda *validate()*.

```
public boolean isValid ();
public void validate();
```

Spre exemplu:

```
import java.awt.*;
import javax.swing.*;

public class DisplayFrameGridLayout {
  public static void main(String[] args) {
    JFrame f = new JFrame("Dispunere cu GridLayout");
    Container cp = f.getContentPane();
    cp.setLayout(new GridLayout(3, 2));
    cp.add(new JButton("1"));
    cp.add(new JButton("2"));
    cp.add(new JButton("3"));
    cp.add(new JButton("4"));
    cp.add(new JButton("5"));
    cp.add(new JButton("6"));
    f.setSize(300, 200);
    f.setVisible(true);
    // adaugam inca doua butoane ceea ce va determina invalidarea containerului
    // si necesitatea reafisarii lui
    cp.add(new JButton("7"));
    cp.add(new JButton("8"));
    if (!cp.isValid())
    // reafisam explicit containerul prin re-validarea lui
      cp.validate();
  }
}
```

Cu rezultatul:




**Evenimente şi „ascultători”**

După cum am menţionat într-unul din paragrafele anterioare, interacţiunile interfeţelor utilizator grafice se bazează pe paradigma evenimentelor. Anumite evenimente sunt de nivel scăzut cum ar fi apăsarea şi eliberarea unei taste, mutarea cursorului mouse-lui sau apăsarea unui buton al acestuia, iar altele sunt de nivel înalt cum ar fi selectarea unei opţiuni dintr-un meniu, apăsarea unui “buton” (componenta grafică) sau introducere de text într-un câmp. Evenimentele de nivel înalt implică de fapt mai multe de nivel mai scăzut. De exemplu introducerea de text într-un câmp implică mutarea cursorului mouse-ului, apăsarea butonului mouse-ului şi apăsarea şi eliberarea mai multor taste.

Iată câteva dintre categoriile de evenimente întâlnite:

| **Categoria de evenimente** | **Descriere** |
|---|---|
| ***Key event*** | Apăsarea şi eliberarea tastelor |

| | |
|---|---|
| ***Mouse event*** | Apăsarea şi eliberarea butoanelor mouse-ului<br>Drag & drop |
| ***Component event*** | Ascunderea, afişarea, redimensionarea sau mutarea unei componente |
| ***Container event*** | Adăugarea sau îndepărtarea unei componente dintr-un container |
| ***Window event*** | Deschiderea, închiderea, minimizarea (iconificarea), reconstituirea (deiconificarea), activarea, dezactivarea unei ferestre |
| ***Focus event*** | Preluarea sau pierderea focus-ului de către o componentă |
| ***Action event*** | Eveniment de nivel înalt indicând o acțiune definită la nivelul componentei (de exemplu un buton poate fi apăsat, un checkbox poate fi selectat, apăsarea tastei ENTER/RETURN pentru un câmp de text) |
| ***Item event*** | Eveniment de nivel înalt care are loc atunci când un utilizator selectează un checkbox, buton radio sau opțiune dintr-o listă |
| ***Document event*** | Generat de către un obiect *TextComponent* atunci când îi este modificat conținutul |

Un aspect important de reținut este faptul instanțele de tip *Component* reprezintă surse de evenimente, mai exact evenimentele au loc în contextul componentelor.

Majoritatea evenimentelor sunt modelate având la bază clasa abstractă *java.awt.AWTEvent*, care ea însăşi este o subclasă a *java.util.EventObject*. Referința obiectului sursă al unui eveniment poate fi obținută prin folosirea metodei *getSource()* a obiectelor de tip *Event*.

```
public object getSource () ;
```

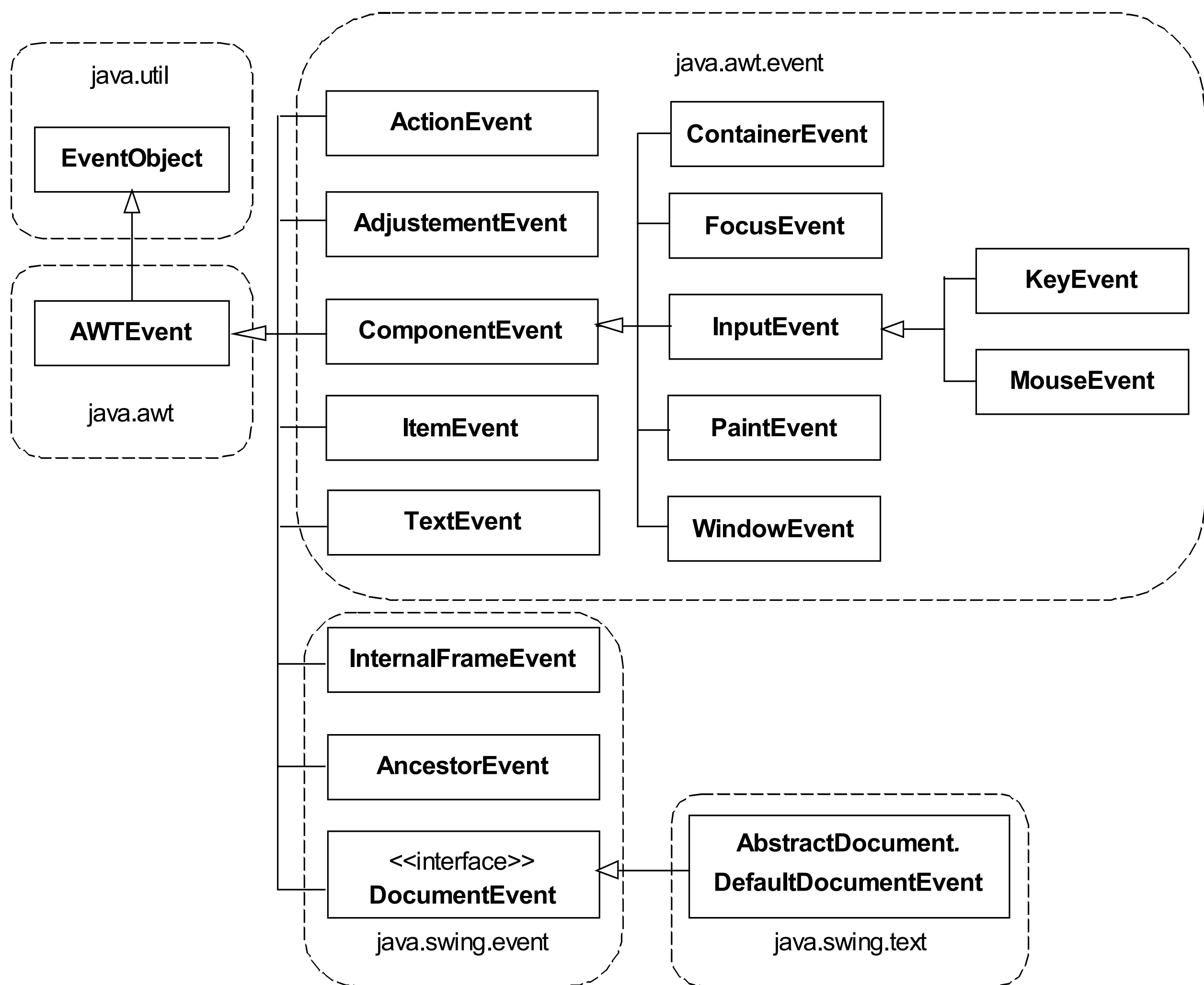



Figura 2-6 O parte din clasele pentru evenimetele Java

Evenimentele generate de componente sunt preluate şi tratate de obiecte specifice care trebuie să se conformeze anumitor interfețe. În Java sunt definite interfețe pentru fiecare categorie de evenimente, iar obiectele "co-interesate" în tratarea lor trebuie, prin urmare, să implementeze aceste interfețe, adică sunt obligate să dețină metodele care vor fi apelate la apariţia respectivelor evenimente.

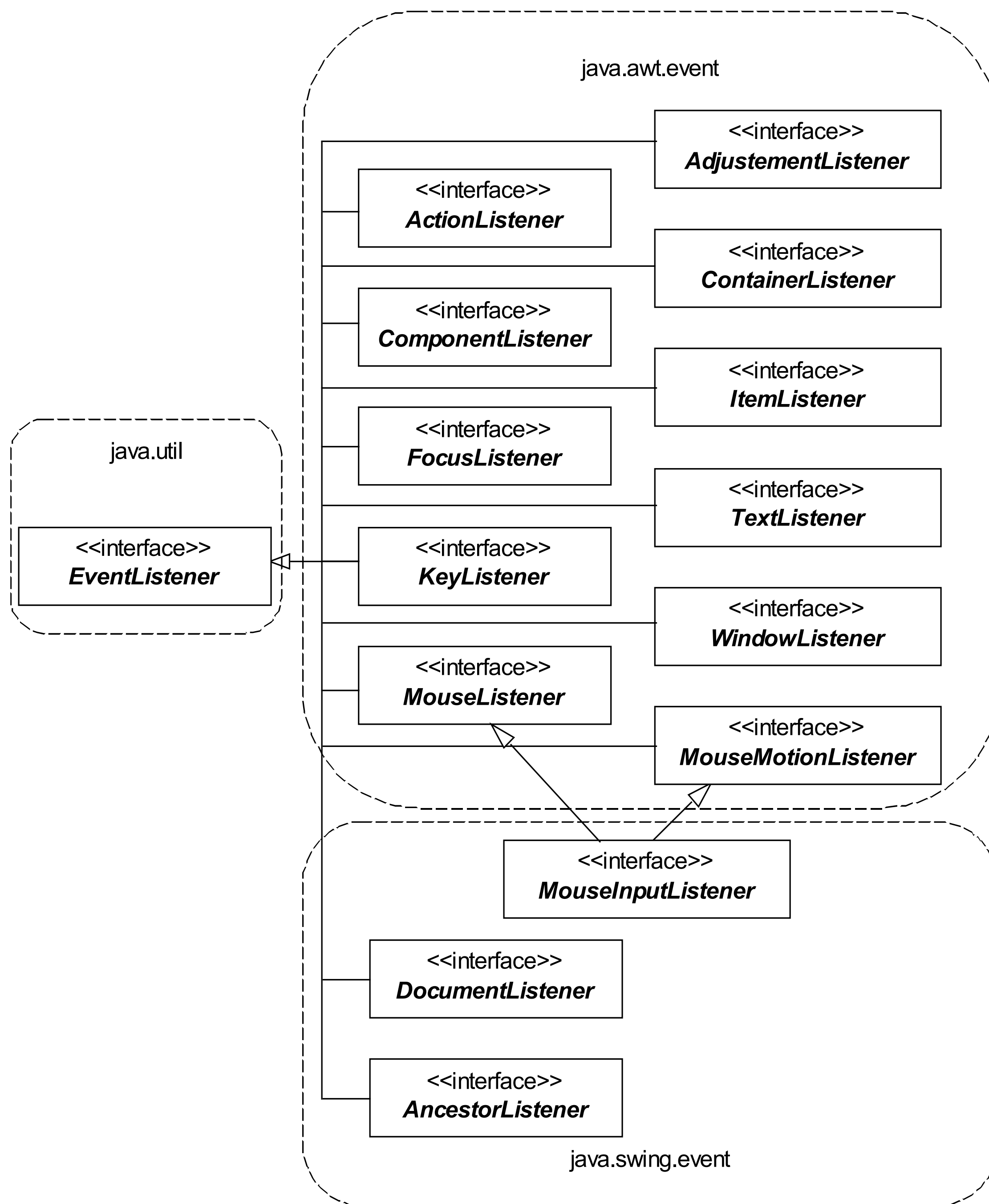



Figura 2-7 Interfața *EventListener* cu extensiile sale

Astfel, analizând codul următor:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class Intermitent extends JFrame {
  public Intermitent () {
    super("Intermitenta");
    JButton button = new JButton("Schimba");
    // Stabilesc fond rosu si font albastru
    button.setBackground(Color.red);
    button.setForeground(Color.blue);
```

```
    // Creez un obiect capabil sa trateze evenimente
    Comutator control = new Comutator();
    // Inregistrez obiectul creat anterior ca destinatie pentru evenimentele
    // generate de buton
    button.addActionListener(control);
    Container cp = this.getContentPane();
    // Adaug butonul la containerul JFrame prin intermediul content pane-ului sau
    cp.add(button, BorderLayout.CENTER);
    this.setSize(300, 200);
    // Afisez JFrame-ul
    this.setVisible(true);
  }
}

// Creez obiectul care receptioneaza si trateaza evenimentele butonului
// si fiindca este inregistrat prin metoda addActionListener, va
// receptiona evenimente din categoria ActionEvent, deci va fi obligat
// sa implementeze interfata ActionListener
class Comutator implements ActionListener{
// Interfata ActionListener obliga implementarea
// metodei actionPerformed(ActionEvent)
  public void actionPerformed(ActionEvent e){
// Tratarea evenimentului inseamna schimbarea colorii fondului
// cu cea a fontului, si invers
    Component sursa = (Component)e.getSource();
    Color prima = sursa.getForeground();
    sursa.setForeground(sursa.getBackground());
    sursa.setBackground(prima);
  }
}

// Clasa TestEvent va "produce" fereastra/cadrul Intermitent
// prin instantiere
public class TestEvent {
  public static void main(String[] args) {
    Intermitent i = new Intermitent();
  }
}
```

putem deduce că:

- un obiect sursă de evenimente (o componentă) are nevoie de un obiect care să îi recepţioneze şi să-i trateze evenimentele, obiect desemnat printr-o metodă specifică cum ar fi *addActionListener* pentru evenimente din categoria *ActionEvent*, sau alte metode specifice (*AddAncestorListener(AncestorListener), addComponentListener(ComponentListener), addFocusListener(FocusListener), addItemListener(ItemListener), addKeyListener(KeyListener), addMouseListener(MouseListener)*) pentru alte tipuri de evenimente;
- obiectul receptor trebuie să implementeze o interfaţă specifică cum ar fi *ActionListener* specifică obiectelor pe care le poate recepţiona – *ActionEvent*, instanţă care presupune implementarea unei metode care va fi de fapt apelată la producerea respectivului eveniment – *actionPerformed*.

Modul în care colaborează obiectele implicate în exemplul de mai sus poate fi redat astfel:

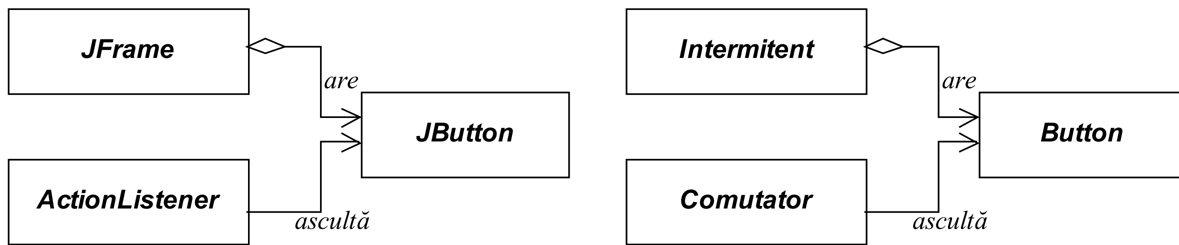



Figura 2-8 Colaborare bazată pe evenimente

Am menţionat, la un moment dat, că închiderea frame-urilor afişate nu implică automat şi încheirea execuţiei procesului respectiv. Încheirea execuţiei unui process se poate realiza prin invocarea metodei *System.exit()*. Când însă ar trebui executată această metodă ? Răspunsul logic ar fi la închiderea ferestrei.. Evenimentele specifice ferestrelor vor fi preluate de obiecte ce corespund interfeţei *WindowListener*. Această interfaţă este ceva mai complicată decât *ActionListener* pentru că presupune mai multe metode:

`public void windowActivated(WindowEvent event)`

Invocată atunci când fereastra respectivă va prelua controlul şi va recepţiona acţiunile utilizatorului de la tastatură.

`public void windowClosing(WindowEvent event)`

Invocată atunci când utilizatorul încearcă în mod explicit închiderea ferestrei.

`public void windowClosed(WindowEvent event)`

Invocată în momentul în care fereastra a fost închisă.

`public void windowDeactivated(WindowEvent event)`

Invocată atunci când fereastra nu mai este fereastra activă, adică nu va mai recepţiona evenimentele determinate de acţiunile de la tastatură.

`public void windowIconified(WindowEvent event)`

Invocată atunci când fereastra este minimizată.

`public void windowDeiconified(WindowEvent event)`

Invocată atunci când fereastra este restaurată la forma normală.

`public void windowOpened(WindowEvent event)`

Invocată prima dată când fereastra devine vizibilă.

Prin urmare dacă JFrame-ul *Intermitent* îşi va înregistra prin metoda *addWindowListener(WindowListener)* un obiect care va recepţiona evenimentele specifice ferestrelor şi care va fi obligat astfel să respecte interfaţa *WindowListener*, atunci instrucţiunea cu pricina ar trebui specificată în metoda *windowClosing(WindowEvent)*. Programul iniţial modificându-se astfel:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;


class IntermitentClosing extends JFrame {
  public IntermitentClosing () {
    super("Intermitenta");

// Inregistrez obiectul care va prelua evenimentele specifice ferestrei
```

```
    this.addWindowListener(new Inchizator());

    JButton button = new JButton("Schimba");
    button.setBackground(Color.red);
    button.setForeground(Color.blue);
    button.addActionListener(new ComutatorClosing());
    Container cp = this.getContentPane();
    cp.add(button, BorderLayout.CENTER);
    this.setSize(300, 200);
    this.setVisible(true);
  }
}

class ComutatorClosing implements ActionListener{
  public void actionPerformed(ActionEvent e){
    Component sursa = (Component)e.getSource();
    Color prima = sursa.getForeground();
    sursa.setForeground(sursa.getBackground());
    sursa.setBackground(prima);
  }
}

// definesc obiectul care va trata evenimentele specifice ferstrelor şi care
// va trebui să se conformeze interfeței WindowListener
class Inchizator implements WindowListener{
  public void windowActivated(WindowEvent event){}
// Metoda windowClosing va determina încheierea procesului
  public void windowClosing(WindowEvent event){
    System.exit(0);
  }
  public void windowClosed(WindowEvent event){}
  public void windowDeactivated(WindowEvent event){}
  public void windowIconified(WindowEvent event){}
  public void windowDeiconified(WindowEvent event){}
  public void windowOpened(WindowEvent event){}
}

public class TestEventClosing {
  public static void main(String[] args) {
    IntermitentClosing i = new IntermitentClosing();
  }
}
```

În listingul de mai sus se observă însă că, obligată fiind de interfața *WindowListener*, clasa *Închizător* a specificat toate metodele, însă implementare efectivă a specificat numai pentru *windowClosing()* celelate având specificat un bloc de intrucțiuni gol. Pentru a simplifica implementarea unui *Listener*, Java furnizează o colecție de clase *adapter* abstracte, care implementează interfețele *Listener* cu metode nule. Prin urmare pentru a implementa o clasă *Listener* se poate folosi o astfel de clasă şi se suprascriu numai metodele care prezintă interes.

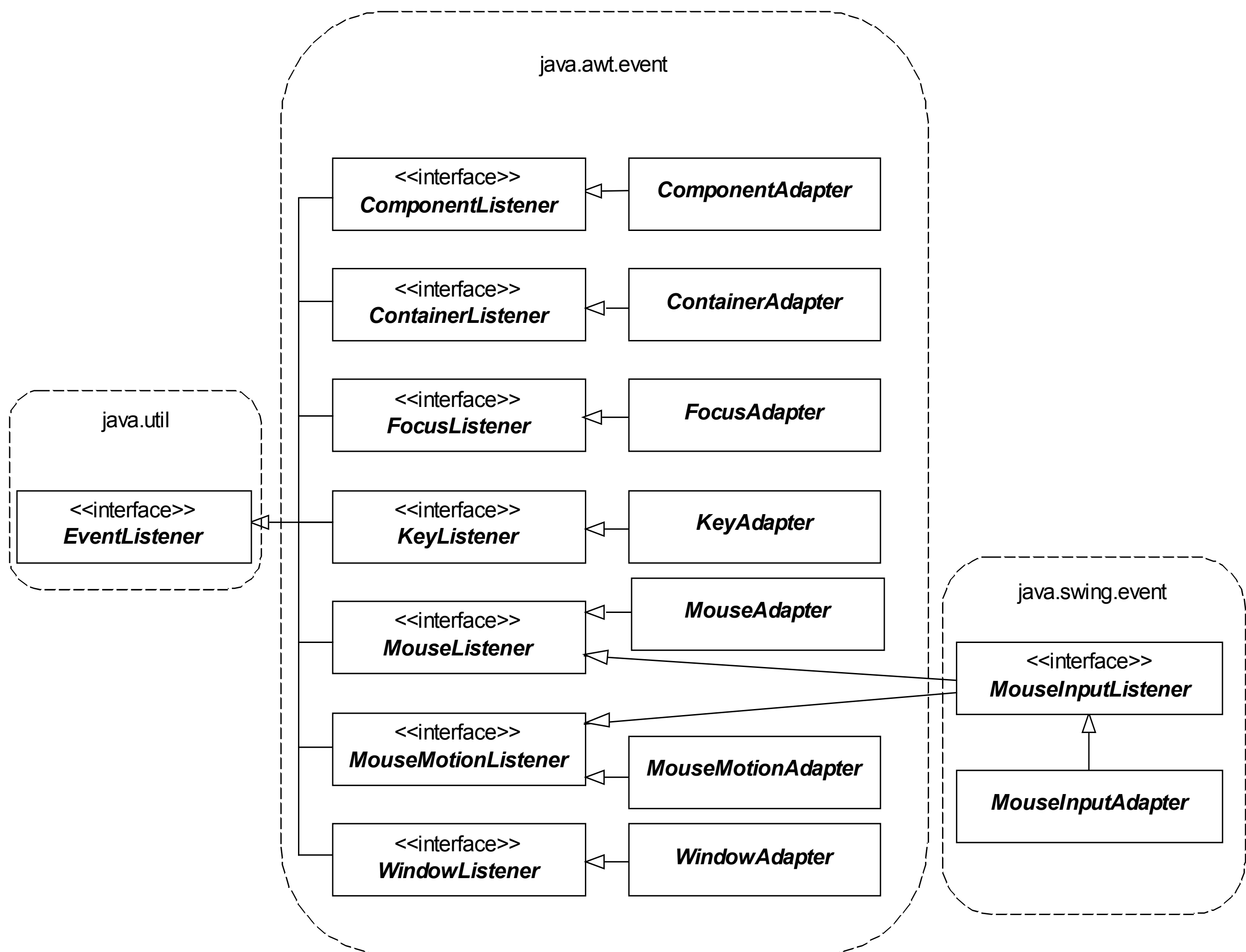



Figura 2-9 O parte din interfețele *Listener* împreună cu clase lor *Adapter*

În acest context clasa *Închizător* din ultimul listing poate fi modificată după cum urmează:

```
class Inchizator extends WindowAdapter{
  public void windowClosing(WindowEvent event){
    System.exit(0);
  }
}
```

O cale mai elegantă de a închide o fereastră este apelarea unei metodei *dispose()* pentru un obiect de tip *Window*. Cum clasa *JFrame* derivă din clasa *Window* înseamnă că pentru clasa *Închizător* care tratează metodele specifice ferestrelor pentri cadrul *JFrame* putem specifica:

```
class Inchizator extends WindowAdapter{
  public void windowClosing(WindowEvent event){
    event.getWindow().dispose();
  }
  public void windowClosed(WindowEvent event){
    System.exit(0);
  }
}
```

Argumentul 0 al metodei *exit()* din clasa *System* indică o închidere normală a aplicației.

### 2.4.3. Construirea interfeţelor grafice – modul de utilizare al principalelor componente

În continuare vom descrie mai în detaliu funcţionalitatea componentelor grafice implicate într-o aplicaţie cu formulare Swing.

#### *2.4.3.1. Rolul layout-urilor şi container-elor în Swing*

În cele ce urmează revenim pentru moment la *componente* şi *container-e*. După cum am evidenţiat mai înainte, un *control* reprezintă un element afişabil pe ecran, un *container* reprezintă o zonă ce grupează fizic (conţine) controale şi/sau eventual alte container-e, iar o *componentă* am putea spune că reprezintă o unitate constructivă pentru interfeţele grafice Java luând forma concretă fie a controalelor fie a containerelor.

Container-ele disponibile prin bibliotecile Swing sunt diferite de cele din biblioteca AWT fiind organizate pe mai multe straturi:

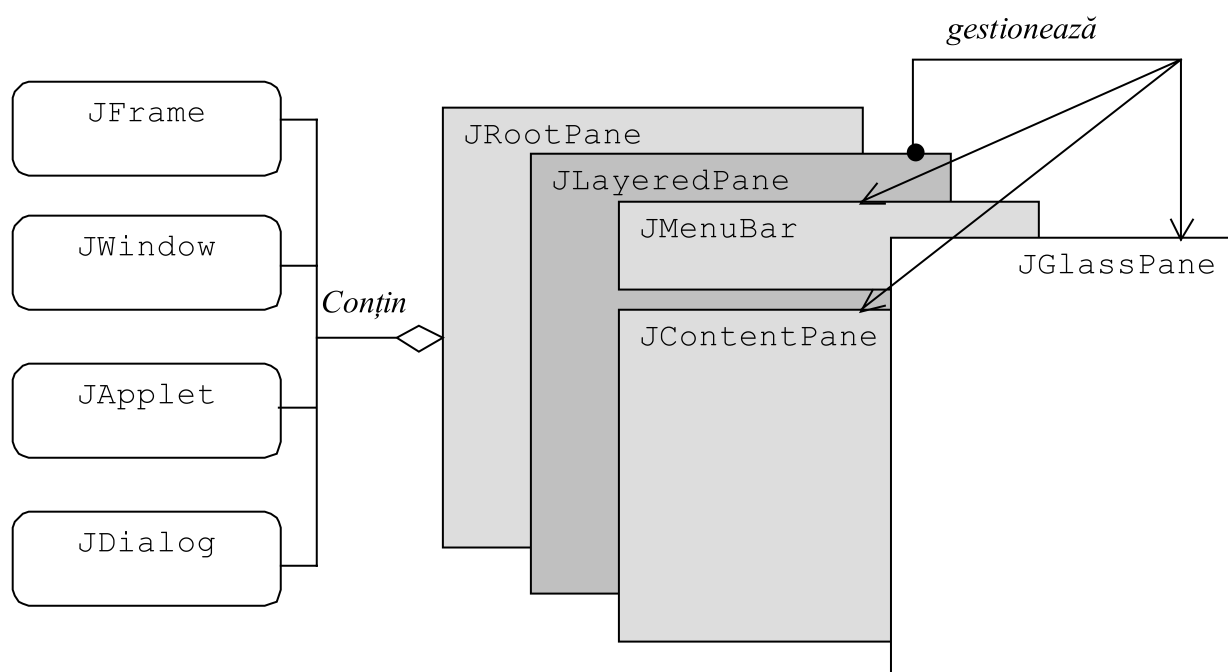



Figura 2-10 Straturile containerelor Swing

Aceste straturi pot fi descrise, pe scurt, astfel:

- *JRootPane* reprezintă structura de date care conţine toate celelalte "panouri" provenind din containerele Swing.
- *JLayeredPane* gestionează "panourile" pentru meniu şi pentru celelalte componente grafice. Prin urmare acest container conţine altele două: *JMenuBar* şi *JContentPane*. De asemenea, tot la acest nivel, apare şi noţiunea *z-order*, adică ordinea în care sunt stratificate componentele (care componente apar deasupra altora).
- *JContentPane* reprezintă un container Swing în care vor fi adăugate toate celelalte componente (controale sau alte container-e) şi la nivelul căruia este stabilită "politica" de dispunere a acestora (noţiunea de *layout manager*).

- *JMenuBar* va fi obiectul prin intermediul căruia sunt gestionate eventualele meniuri bară ataşate applet-ului sau frame-ului respective.
- *JGlassPane* reprezintă un container transparent care este aşezat peste toate celelalte componente şi container-e şi permite interceptarea evenimentelor legate mouse, precum şi posibilitatea schiţării unor obiecte grafice peste întegul conţinut fără să afecteze vreo componentă existentă.

Referinţele acestor obiecte pot fi obţinute astfel:

```
JRootPane rp = getRootPane();
JLayeredPane lp = getLayeredPane();
JMenuBar mb = getJMenuBar();
Container cp = getContentPane();
Component gp = getGlassPane();
```

Acest mod de structurare a conţinutului inerfeţelor grafice este comun tuturor container-elor top-level, adică cele care nu sunt subincluse în altele şi care afişează ferestrele principale: *JFrame, JWindow, JApplet, JDialog*. Iată câteva din caracteristicile esenţiale pentru aceste container-e top-level:

- *JFrame* reprezintă o fereastră care conţine implicit: o bară de titlu, o bară de meniu, un chenar (margine – numită *inset*) şi care poate fi închisă sau minimizată (iconificată).
- *JApplet* este o subclasă derivată din *JPanel* care reprezintă un container generic care se regăseşte întotdeauna în alt container însă nu are proprietatea de a "pluti" liber pe un desktop. Obiectele create din *JApplet* (de altfel la fel ca şi cele din clasa *Applet* din AWT) pot fi afişate direct într-o pagină web obişnuită. Ele conţin o serie de metode care iniţializează şi controlează execuţia lor în pagina Web (sau în AppletViewer în faza de test) *start(), init(), stop(), destroy()*, precum şi o serie de metode pentru a lucra cu informaţii multimedia: sunete şi imagini.
- *JWindow* reprezintă un container gol – o fereastră lipsită implicit de conţinut grafic, neavând nici măcar chenar. O fereastră nu este folosită direct , ci prin intermediul subclaselor specializate *JFrame* şi *JDialog*.

**Z-order**

Suprapunerea controalelor sau componentelor nu reprezintă o noutate şi a fost implementată şi în interfeţele grafice realizate cu biblioteca Swing. Însă atunci când se produce un astfel de fenoomen devine importantă ordinea componentelor nu pe axa *x* sau *y*, ci pe axa *z*, de aici şi noţiunea *z-order*. Prin urmare ordinea pe axa z – z-order – reprezintă plasamentul obiectelor pe ecran unele deasupra altora, lucru valabil bineînţeles pentru componentele "uşoare".

Dispunerea suprapusă a componentelor este dependentă de ordinea în care sunt adăugate în container (de exemplu în JContentPane). Iată un exemplu în acest sens:

```
import java.awt.*;
import javax.swing.*;

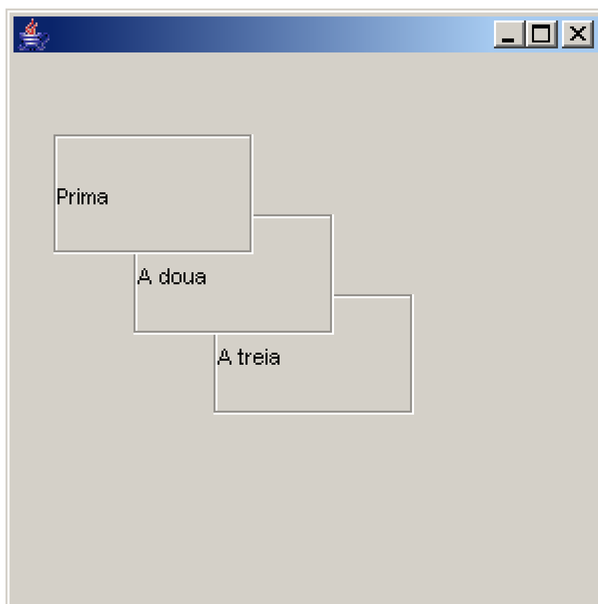
public class ExempluZOrder extends JFrame {
  JLabel label1 = new JLabel("Prima");
```

```
    JLabel label2 = new JLabel("A doua");
    JLabel label3 = new JLabel("A treia");
    public ExempluZOrder() {
     label1.setOpaque(false);
     label2.setOpaque(false);
     label3.setOpaque(false);

     Container contentPane = getContentPane();
     contentPane.setLayout(null);
     contentPane.add(label1);
     contentPane.add(label2);
     contentPane.add(label3);

     label1.setBounds(20, 40, 100, 60);
     label1.setBorder(BorderFactory.createEtchedBorder());
     label1.setOpaque(true);
     label2.setBounds(60, 80, 100, 60);
     label2.setBorder(BorderFactory.createEtchedBorder());
     label2.setOpaque(true);
     label3.setBounds(100, 120, 100, 60);
     label3.setBorder(BorderFactory.createEtchedBorder());
     label3.setOpaque(true);
     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     setSize(300, 300);
    }
    public static void main(String[] args){
       new ExempluZOrder().show();
    }
   }
```

Efectul:




Un rol cheie în exemplul de mai sus îl joacă şi metoda *setOpaque()* a cărei menire este să afişeze în mod transparent (cu argumentul *false*) sau în mod opac (cu argumentul *true*) componenta la care se face referire.

**Folosirea panourilor suprapuse**

După cum am văzut într-un paragraf anterior, container-ului *JRootPane* conține un altul cu rol esențial în modul de prezentare a applet-urilor şi aplicațiilor, şi anume *JLayeredPane*. Am remarcat, de asemenea, că la nivel de bază acest container conține bara de meniuri şi componenta *content pane*.

*JLayeredPane* împarte modul în care suprapune diferitele componente pe care le conține în mai multe straturi:

- **DEFAULT_LAYER** – este stratul cel mai de jos (standard) pe care sunt aşezate majoritatea componentelor;
- **PALLETE_LAYER** – este stratul aşezat deasupra celui "default" în care rezidă de obicei barele de instrumente flotante;
- **MODAL_LAYER** – este stratul folosit de regulă pentru căsuțele de dialog modale;
- **POPUP_LAYER** – este aşezat deasupra stratului pentru dialoguri;
- **DRAG_LAYER** – atunci când o componentă este "plimbată" (drag) se recomandă asignarea ei acestui strat pentru a se poziționa deasupra tutror celorlalte componente din container.

Aceste nume desemnează de fapt constantele care se regăsesc sub formă de membri de tip *Integer* ai clasei *JLayeredPane*, şi ale căror valori de tip *int* pot fi obținute printr-o instrucțiune de genul *JLayeredPane.PALLETE_LAYER.intValue()*.

Metodele esențiale folosite pentru poziționarea sau repoziționarea componentelor în aceste straturi aparțin tot obiectelor *JLayeredPane* şi se referă la:

- *moveToFront(Component c)* – duce componenta specificată deasupra tuturor celorlalte din stratul ei curent
- *moveToBack(Component c)* - duce componenta specificată în spatele tuturor celorlalte din stratul ei curent
- *setPosition(Component c, int layer)* – mută componenta specificată pe alt nivel în stratul ei curent
- *setLayer(Component c, int layer)* – stabileşte componenta specificată *layer*-ul specificat prin al doilea parametru.

Pentru exemplificarea modului în care se suprapun aceste straturi iată exemplul următor:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.JApplet;

public class ExempluJLayeredPane extends JFrame {
  JLayeredPane jlpane = new JLayeredPane();
  JLabel label1 = new JLabel(" L1 - Stratul Content");
  JLabel label2 = new JLabel(" L2 - Stratul Default");
  JLabel label3 = new JLabel(" L3 - Stratul Palette");
  JLabel label4 = new JLabel(" L4 - Stratul Modal");
  JLabel label5 = new JLabel(" L5 - Stratul Popup");
  JLabel label6 = new JLabel(" L6 - Stratul Drag");

  public ExempluJLayeredPane() {
```

```
        setContentPane(jlpane);

        label1.setBounds(20, 0, 120, 60);
        label2.setBounds(30, 40, 120, 60);
        label3.setBounds(40, 80, 120, 60);
        label4.setBounds(50, 120, 120, 60);
        label5.setBounds(60, 160, 120, 60);
        label6.setBounds(70, 200, 120, 60);

        label1.setBorder(BorderFactory.createEtchedBorder());
        label2.setBorder(BorderFactory.createEtchedBorder());
        label3.setBorder(BorderFactory.createEtchedBorder());
        label4.setBorder(BorderFactory.createEtchedBorder());
        label5.setBorder(BorderFactory.createEtchedBorder());
        label6.setBorder(BorderFactory.createEtchedBorder());

        jlpane.setLayer(label1,JLayeredPane.FRAME_CONTENT_LAYER.intValue());
        jlpane.setLayer(label2,JLayeredPane.DEFAULT_LAYER.intValue());
        jlpane.setLayer(label3,JLayeredPane.PALETTE_LAYER.intValue());
        jlpane.setLayer(label4,JLayeredPane.MODAL_LAYER.intValue());
        jlpane.setLayer(label5,JLayeredPane.POPUP_LAYER.intValue());
        jlpane.setLayer(label6,JLayeredPane.DRAG_LAYER.intValue());

        jlpane.add(label1);
        jlpane.add(label2);
        jlpane.add(label3);
        jlpane.add(label4);
        jlpane.add(label5);
        jlpane.add(label6);

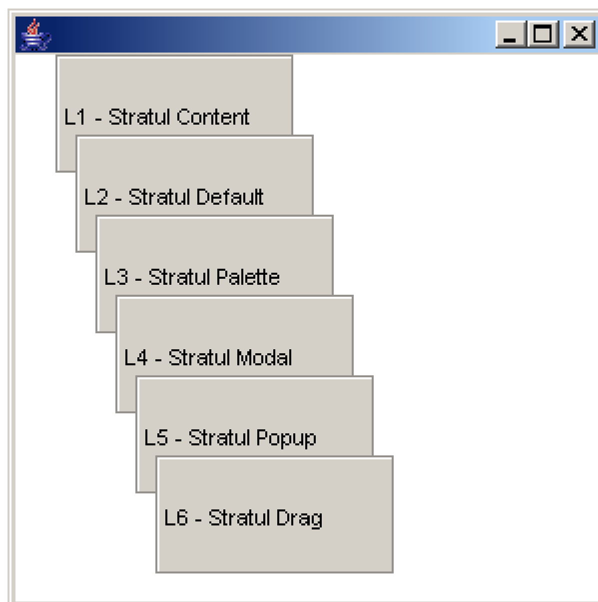
        label1.setOpaque(true);
        label2.setOpaque(true);
        label3.setOpaque(true);
        label4.setOpaque(true);
        label5.setOpaque(true);
        label6.setOpaque(true);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 300);

    }

    public static void main(String[] args) throws Exception{
        UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
        new ExempluJLayeredPane().show();
    }
}
```

Efectul va fi:




**JTabbedPane – o altfel de stratificare**

Panourile care au aspectul unor pagini etichetate suprapuse se pot obține folosind clasa *JTabbedPane* din Swing. Pentru a obține cel mai simplu panou de acest fel avem nevoie cel puțin de metodele:

- Constructorul *JTabbedPane*();
- Metoda *addTab(String titlu, Component componentă)* prin care se obțin paginile suprapuse ale panoului;

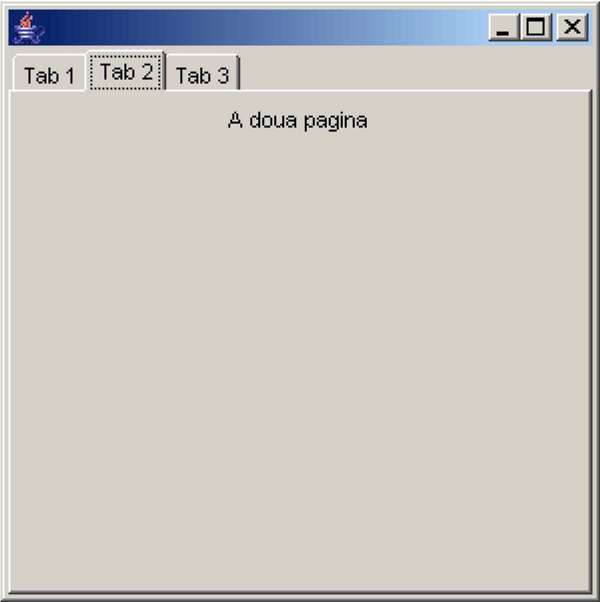
Iată un exemplu simplificat de realizare a unui astfel de panou ale cărui pagini sunt obținute din obiectul *JPanel* care reprezintă structura cea mai simplă de a grupa componente grafice (în special controale):

```
import java.awt.*;
import javax.swing.*;
import javax.swing.JApplet;

public class ExempluJTabbedPane extends JFrame {
  public ExempluJTabbedPane() {
    Container contentPane = getContentPane();
    JTabbedPane jtab = new JTabbedPane();
    JPanel jpanel1 = new JPanel();
    JPanel jpanel2 = new JPanel();
    JPanel jpanel3 = new JPanel();
    jpanel1.add(new JLabel(" Prima pagina "));
    jpanel2.add(new JLabel(" A doua pagina "));
    jpanel3.add(new JLabel(" A treia pagina "));
    jtab.addTab("Tab 1", jpanel1);
    jtab.addTab("Tab 2", jpanel2);
    jtab.addTab("Tab 3", jpanel3);
    contentPane.setLayout(new BorderLayout());
    contentPane.add(jtab);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(300, 300);
  }
    public static void main(String[] args) throws Exception{
      UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
      new ExempluJTabbedPane().show();
```

```
        }
    }
```

Vom obține următorul rezultat:




### 2.4.3.2. *Modul de utilizare şi comportamentul principalelor componente grafice Swing*

**Labels – etichete**

Biblioteca Swing aduce, odată cu Java 2, o componentă mai specializată decât clasicul java.awt.*Label* pentru afişarea simplelor texte şi anume *javax.swing.JLabel* care este derivată direct din *javax.swing.JComponent*:

```
java.lang.Object
| ___ java.awt.Component
        | ___ java.awt.Container
                | ___ javax.swing.JComponent
                        | ___ javax.swing.Jlabel
```

Principalii constructori prin care se poate instanția o etichetă folosind clasa JLabel sunt:

| ***Constructor*** | ***Acțiune*** |
|---|---|
| **JLABEL()** | Construieşte un obiect JLabel simplu fără imagine sau text |
| JLabel(String text) | Construieşte un JLabel ce afişează textul specificat |
| JLabel(String text, int horizontalAlignament) | Construieşte un JLabel ce afişează textul aliniat corespunzător specificației |
| JLabel(Icon image) | Construieşte un JLabel ce afişează imaginea specificată |
| JLabel(String text, Icon icon, int horizontalAlignment) | Construieşte un JLabel care afişează un text şi un obiect Icon |

Principalele metode folosite în manipularea obiectelor JLabel:

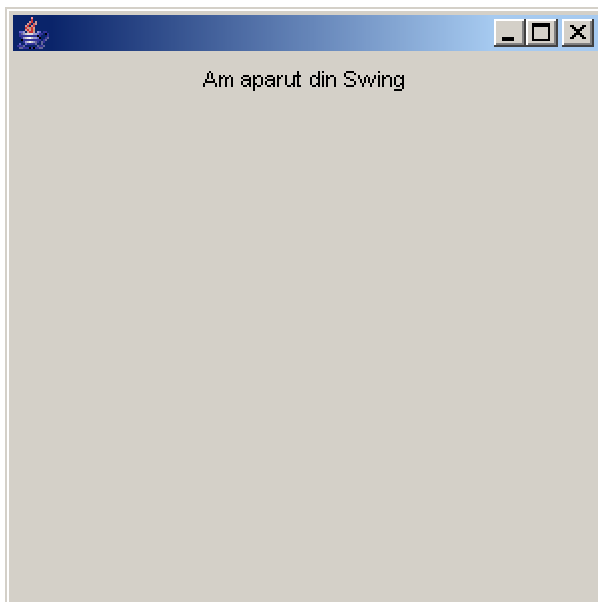
| ***Metoda*** | ***Acțiune*** |
|---|---|
| String getText() | Obține Stringul afişat de către obiectul JLabel |
| void setText(String text) | Stabileşte printr-un String textul ce va fi afişat |
| int getHorizontalAlignment() | Obține un întreg care indică modul de aliniere a conținutului etichetei relativ la axa x |
| int getVerticalAlignment() | Obține un întreg care indică modul de aliniere a conținutului etichetei relativ la axa y |
| Icon getIcon() | Obține obiectul Icon afişat de către obiectul JLabel |
| void setIcon(Icon icon) | Stabileşte obiectul Icon afişată de componenta JLabel |
| Component getLabelFor() | Obține componenta care este etichetată folosind obiectul JLabel |
| void setLabelFor(Component c) | Stabileşte componenta care va fi etichetată cu obiectul JLabel |
| void setHorizontalAlignment(int alignament) | Stabileşte prin întregul specificat modul de aliniere relativ la axa x |
| void setVerticalAlignment(int alignament) | Stabileşte prin întregul specificat modul de aliniere relativ la axa y |

Pentru a obține un obiect JLabel care afişează un text simplu putem construi o astfel de etichetă ca în exemplul următor:

```
import java.awt.*;
import javax.swing.*;

public class ExempluJLabel extends JFrame{
  public ExempluJLabel(){
    Container cp = getContentPane();
    JLabel jlabel = new JLabel("Am aparut din Swing");
    cp.setLayout(new FlowLayout());
    cp.add(jlabel);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(300, 300);
  }
    public static void main(String[] args) throws Exception{
        UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
        new ExempluJLabel().show();
    }
}
```

Iată rezultatul:




**Câmpuri pentru text JTextField**

Biblioteca SWING introduce, pentru a gestiona câmpurile care pot primi text din partea utilizatorilor, componenta *JTextField* ce lucrează asemănător cu java.awt.*TextField* (dar care nu este derivată din aceasta).

java.lang.Object
| ___ java.awt.Component
    | ___ java.awt.Container
        | ___ javax.swing.JComponent
            | ___ javax.swing.text.JTextComponent
                | ___ javax.swing.JTextField

Constructorii principali ai componentei *JTextField* sunt următorii:

| ***Constructor*** | ***Acțiune*** |
|---|---|
| **JTEXTFIELD()** | Creează un nou câmp gol |
| JTextField (int columns) | Creează un nou obiect JTextField fără conținut de lungimea indicată (în coloane) |
| JTextField(String text) | Creează un nou obiect JTextField conținând textul specificat |
| JTextField(String text, int columns) | Creează un nou obiect JTextField conținând textul specificat şi având (afişând) lungimea indicată (în coloane) |

Principalele metode ale componentei JTextField sunt următoarele:

| ***Metoda*** | ***Acțiune*** |
|---|---|

| | |
|---|---|
| String getText() | Obține String-ul conținut în câmp |
| void setText(String text) | Stabileşte printr-un String textul ce va fi conținut de către obiectul JTextField |
| int getHorizontalAlignment() | Obține un întreg care indică modul de aliniere relativ la axa x |
| void setHorizontalAlignment(int alignament) | Stabileşte prin întregul specificat modul de aliniere relativ la axa x |
| Dimension getPreferredSize() | Obține dimensiunile (lățime, lungime printr-un obiect de tip Dimension) preferențiale |
| int getColumns() | Obține numărul de coloane pe care este afişat textul |
| void setColumns(int columns) | Stabileşte numărul de coloane pe care este afişat textul |

Pentru a obține un obiect *JTextField* putem proceda ca în exemplul următor:

```
public class ExempluJTextField extends JFrame{
   JTextField text = new JTextField();
   public ExempluJTextField() {
      Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      cp.add(text);
      text.setText("Text cu JTextField");
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setSize(300, 300);
   public static void main(String[] args) throws Exception{
      UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
      new ExempluJTextField().show();
   }
}
```

Iată rezultatul:




Obiectele *JtextField* generează evenimente legate de apăsarea tastatelor şi un *ActionEvent* atunci când utilizatorul apasă tasta *Enter/Return* pentru a încheia introducerea textului şi eventual pentru a-l valida. Codul pentru implementarea modelului de evenimente legate de această componentă ar putea fi:

```
text.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e){
  // secvența de cod ce trebuie executată la apariția ecenimentului sau
  // apel la o eventuală metodă din cadrul părinte în care să fie tratat explicit acest eveniment
    System.out.println("Textul introdus " + e.getActionCommand());
  }
});
```

care poate fi adăugat în metoda *init()* a clasei *ExempluTextField* (eventual la sfârşit, sau oricum după instanțierea clsei *JTextField*).

**Butoane – AbstractButton şi JButton**

În Swing, toate butoanele sunt derivate din clasa *AbstractButton* care furnizează baza pe care sunt construite toate clasele pentru butoane din *Java*. Iată ierarhia de moştenire pentru această clasă:

```
java.lang.Object
| ___ java.awt.Component
        | ___ java.awt.Container
                | ___ javax.swing.JComponent
                        | ___ javax.swing.AbstractButton
```

În tabelul următor sunt prezentate câteva dintre cele mai utile câmpuri şi metode ale acestei clase:

| ***Metoda*** | ***Acțiune*** |
|---|---|
| boolean isSelected() | Obține starea butonului |
| int getHorizontalAlignment() | Obține un întreg care indică modul de aliniere relativ la axa x |
| String getText() | Obține String-ul afişat de buton |
| void addActionListener(ActionListener l) | Înregistrează un action listener |
| void addChangeListener(ChangeListener l) | Înregistrează un change listenre |
| void addItemListener(ItemListener l) | Înregistrează un item listener |
| void getVerticalAlignment() | |
| void removeActionListener(), removeChangeListener(), reomoveItemListener() | Renunță la action/change/item listener-ul înregistrat |
| void setEnabled(boolean b) | Activează/dezactivează butonul |
| void setHorizontalAlignment(int alignament) | Stabileşte prin întregul specificat modul de aliniere relativ la axa x |
| void setSelected(boolean b) | Stabileşte starea butonului (selectat/deselectat) |
| void setText() | Stabileşte textul afişat de către buton |
| void setText(String text) | Stabileşte printr-un String textul ce va fi conținut de către obiectul JtextField |
| Icon getIcon() | |

| | |
|---|---|
| void setIcon(Icon defaultIcon) | |

Clasa Swing furnizată pentru implementarea butoanelor de comandă este clasa *JButton*, derivată din clasa *AbstractButton*:

```
java.lang.Object
| ___ java.awt.Component
        | ___ java.awt.Container
                | ___ javax.swing.JComponent
                        | ___ javax.swing.AbstractButton
                                | ___ javax.swing.JButton
```

Față de controlul omolog din biblioteca *awt*, componenta *JButton* include şi câteva caracteristici noi cum ar fi:

- Poate avea un *tooltip* prin metoda *setToolTipText()*;
- Pentru a simula din cod acțiunea de "apăsare" a butonului există metoda *doClick*();

Principalii constructori pentru clasa JButton sunt următorii:

| ***Constructor*** | ***Acțiune*** |
|---|---|
| **JBUTTON()** | Construieşte un obiect JButton simplu fără imagine sau text |
| JButton(String text) | Construieşte un JButton ce afişează textul specificat |
| JButton(Icon image) | Construieşte un JButton ce afişează imaginea specificată |
| JButton(String text, Icon icon) | Construieşte un JButton care afişează un text şi un obiect Icon |

Putem obține un JButton ca în exemplul următor:

```
public class ExempluJButton extends JFrame{
  JButton button = new JButton("Apasa");
  JTextField text = new JTextField(30);
    public ExempluJButton() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(button);
        cp.add(text);

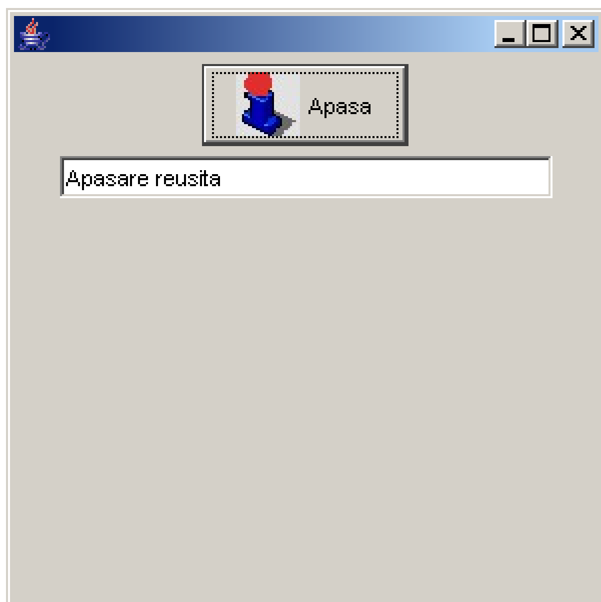
        button.addActionListener(new ActionListener(){
          public void actionPerformed(ActionEvent e){
            text.setText("Apasare reusita");
          }
        });
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 300);
    }
    public static void main(String[] args) throws Exception{
        UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
        new ExempluJButton().show();
    }
}
```

Se poate observa că în metoda *init(),* după instanțierea clasei JButton, este apelată la un moment dat metoda *addActionListener()* pentru înregistrarea unui listener care să trateze evenimentele acestuia.

De asemenea, un JButton poate afişa inclusiv o pictogramă dacă este construit astfel:

```
JButton button = new JButton("Apasa", new ImageIcon("D: \\ProCurs\\img\\jpg\\5.jpg"));
```

Rezultatul:




**Butoane checkbox - JCheckBox**

După cum se ştie, un checkbox reprezintă un control grafic care oferă posibilitatea selectării unei valori booleene, însoțit fiind de un text explicativ. Biblioteca Swing oferă pentru obținerea unui astfel de buton clasa *JCheckBox* care anumite avantaje față de clasa similară din AWT, *CheckBox*, şi anume posibilitatea afişării unei imagini însoțitoare.

```
java.lang.Object
| ___ java.awt.Component
        | ___ java.awt.Container
                | ___ javax.swing.JComponent
                        | ___ javax.swing.AbstractButton
                                |___ javax.swing.JToggleButton
                                        | ___ javax.swing.JCheckBox
```

Principalii constructori care pot fi folosiți pentru instanțierea acestei clase sunt:

| ***Constructor*** | ***Acțiune*** |
|---|---|
| **JCHECKBOX()** | |
| JCheckBox(String text) | |
| JCheckBox(String text, Boolean selected) | Construieşte un JCheckBox ce afişează textul specificat şi indică dacă este, sau nu, selectat inițial |
| JCheckBox(Icon image) | |
| JCheckBox(String text, Icon icon) | |

În ce priveşte metodele, manipularea acestor obiecte se face în principal prin *setSelected(), getSelected(), setText(), getText()* etc. moştenite din superclase, în special de la clasa *JabstractButton*.

Crearea un astfel de checkbox poate fi realizată ca în exemplul următor:

```
public class ExempluJCheckBox extends JFrame implements ItemListener{
   JCheckBox check1, check2, check3, check4;
   JTextField text;
   public ExempluJCheckBox() {
      Container contentPane = getContentPane();
      contentPane.setLayout(new FlowLayout());

      check1 = new JCheckBox("Opt 1");
      check2 = new JCheckBox("Opt 2");
      check3 = new JCheckBox("Opt 3");
      check4 = new JCheckBox("Opt 4");

      check1.addItemListener(this);
      check2.addItemListener(this);
      check3.addItemListener(this);
      check4.addItemListener(this);

      contentPane.add(check1);
      contentPane.add(check2);
      contentPane.add(check3);
      contentPane.add(check4);

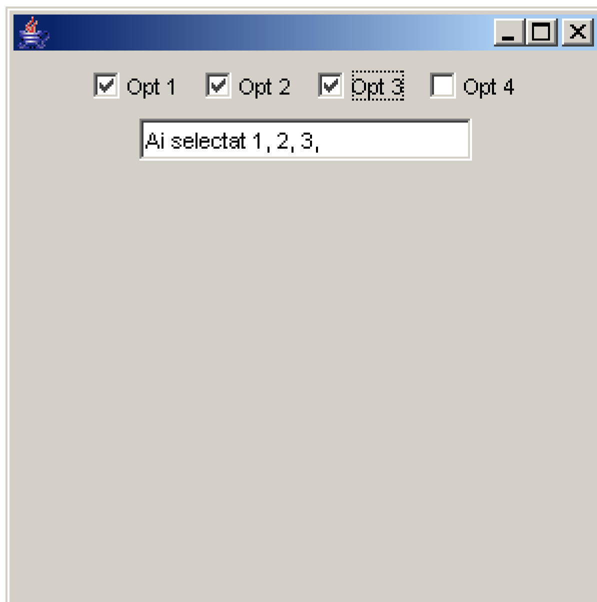
      text = new JTextField(20);

      contentPane.add(text);

      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setSize(300, 300);
   }
   public void itemStateChanged(ItemEvent e)
   {
      String text_ = "Ai selectat ";
      if (check1.isSelected()) text_ += "1, ";
      if (check2.isSelected()) text_ += "2, ";
      if (check3.isSelected()) text_ += "3, ";
      if (check4.isSelected()) text_ += "4, ";
      text.setText(text_);

   }
   public static void main(String[] args) throws Exception{
      UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
      new ExempluJCheckBox().show();
   }
}
```

Rezultatul ar trebui să fie următorul;



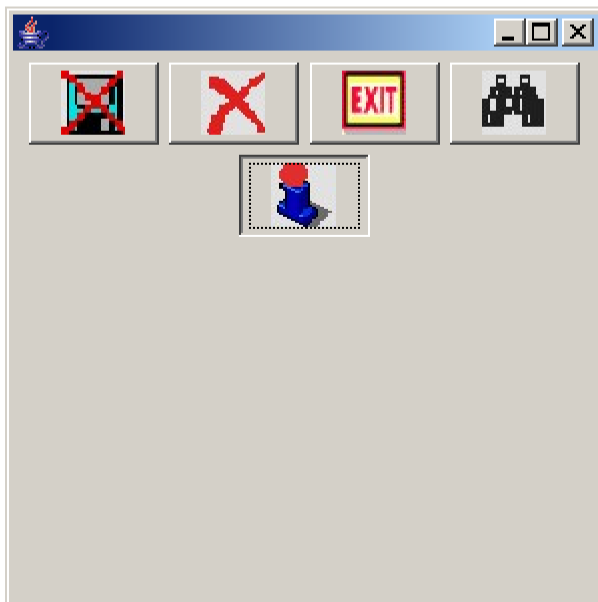

**Grupuri de butoane - JButtonGroup**

Butoanele tip *checkbox* derivă din clasa **JToggleButton**, a cărei principală caracteristică este obținerea de butoane care să poată fi afişate grafic ca selectate sau deselectate şi să fie însoțite de text simplu sau de diferite imagini.

O altă caracteristică importantă a acestor butoane este că pot fi grupate într-o structură comună astfel încât la un moment dat să poată fi selectat numai unul dintre ele. Spre exemplu:

```
public class ExempluJToggleButton extends JFrame{
  public ExempluJToggleButton(){
    Container contentPane = getContentPane();
    ButtonGroup grup = new ButtonGroup();
    JToggleButton [] butoane = new JToggleButton[]{
      new JToggleButton(new ImageIcon("D:\\ProCurs\\img\\jpg\\1.jpg")),
      new JToggleButton(new ImageIcon("D:\\ProCurs\\img\\jpg\\2.jpg")),
      new JToggleButton(new ImageIcon("D:\\ProCurs\\img\\jpg\\3.jpg")),
      new JToggleButton(new ImageIcon("D:\\ProCurs\\img\\jpg\\4.jpg")),
      new JToggleButton(new ImageIcon("D:\\ProCurs\\img\\jpg\\5.jpg"))
    };
    contentPane.setLayout(new FlowLayout());
    for (int i = 0; i < butoane.length; ++i){
      grup.add(butoane[i]);
      contentPane.add(butoane[i]);
    }
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(300, 300);
  }
    public static void main(String[] args) throws Exception{
        UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
        new ExempluJToggleButton().show();
    }
}
```

Rezultatul ar trebui să fie următorul:



**Butoane radio – JRadioButton**

Butoanele radio se folosesc atunci când se doreşte construirea unui grup de butoane de opţiuni însă la un moment dat numai unul dintre ele să fie selectat. La fel ca şi în cazul checkbox-urilor, biblioteca Swing vine cu propria sa clasă *JRadioButton* pentru a obţine butoane de acest fel. Iată ierarhia de moştenire în acest caz:

```
java.lang.Object
| ___ java.awt.Component
        | ___ java.awt.Container
                | ___ javax.swing.JComponent
                        | ___ javax.swing.AbstractButton
                                |___ javax.swing.JToggleButton
                                        | ___ javax.swing.JRadioButton
```

La fel ca în cazul checkkbox-urilor, constructorii clasei JRadioButton pemit crearea de butoane radio însoţite de text, imagini sau text şi imagini, şi specificarea stării de selectat/deselectat:

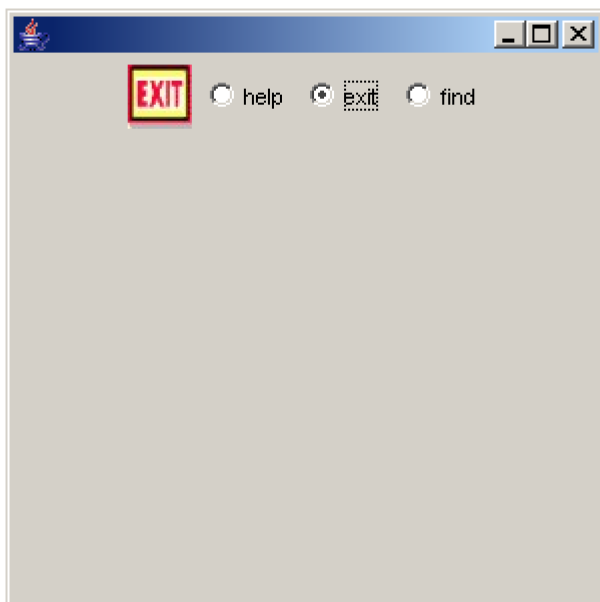
| ***Constructor*** | ***Acţiune*** |
|---|---|
| **JRADIOBUTTON()** | |
| JRadioButton(String text) | |
| JRadioButton(String text, Boolean selected) | Construieşte un JCheckBox ce afişează textul specificat şi indică dacă este, sau nu, selectat iniţial |
| JRadioButton(Icon image) | |
| JRadioButton(String text, Icon icon) | |

Pentru a demonstra modul în care pot fi create şi folosite aceste butoane am construit exemplul următor:

```
public class ExempluJRadioButton extends JFrame{
  JLabel jlabel;
```

```
    ButtonGroup grup;
    JRadioButton [] butoaneRadio;
    JRadioButton b1;
      public ExempluJRadioButton() {
        Container contentPane = getContentPane();
        jlabel = new JLabel();
        jlabel.setIcon(new ImageIcon("D:\\ProCurs\\img\\jpg\\5.jpg"));
        contentPane.add(jlabel);
        grup = new ButtonGroup();
        butoaneRadio = new JRadioButton[]{
          new JRadioButton("help"),
          new JRadioButton("exit"),
          new JRadioButton("find"),
        };
        contentPane.setLayout(new FlowLayout());
        for (int i = 0; i < butoaneRadio.length; ++i){
          grup.add(butoaneRadio[i]);
          contentPane.add(butoaneRadio[i]);
          butoaneRadio[i].addItemListener(new ItemListener(){
            public void itemStateChanged(ItemEvent e){
              refacere_imagine(e);
            }
          });
        }
        butoaneRadio[0].setSelected(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 300);
      }
    public void refacere_imagine(ItemEvent e){
        if (butoaneRadio[0].isSelected())
         jlabel.setIcon(new ImageIcon("D:\\ProCurs\\img\\jpg\\5.jpg"));
        //if (e.getItemSelectable() == butoaneRadio[1])
        if (butoaneRadio[1].isSelected())
         jlabel.setIcon(new ImageIcon("D:\\ProCurs\\img\\jpg\\3.jpg"));
        //if (e.getItemSelectable() == butoaneRadio[2])
        if (butoaneRadio[2].isSelected())
          jlabel.setIcon(new ImageIcon("D:\\POO_2004\\ProCurs\\img\\jpg\\4.jpg"));
    }
      public static void main(String[] args) throws Exception{
        UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
        new ExempluJRadioButton().show();
      }
  }
```

Rezultatul ar trebui să fie următorul:




**Liste şi combobox-uri – JList şi JComboBox**

Listele constituie un gen de controale indispensabile pentru afişarea mai multor opţiuni într-un spaţiu restrâns. Swing oferă în acest sens clasa *JList* a cărei diagramă de moştenire este următoarea:

```
java.lang.Object
| ___ java.awt.Component
        | ___ java.awt.Container
                | ___ javax.swing.JComponent
                        | ___ javax.swing.JList
```

Constructorii care pot fi folosiţi pentru crearea unei liste sunt următorii:

| ***Constructor*** | ***Acţiune*** |
|---|---|
| **JLIST()** | Construieşte o listă simplă |
| JList(ListModel dataModel) | Construieşte un obiect JList care afişează elementele după modelul de date specificat |
| JList(Object [ ] listData) | Construieşte un obiect JList care afişează obiectele din tabloul indicat |
| JList(Vector listData) | Construieşte un obiect JList care afişează elementele din vectorul indicat |

Pentru manipularea la nivel de bază a obiectelor de tip JList găsim următoarele metode:

| ***Metoda*** | ***Acţiune*** |
|---|---|
| void addListSelectionListener (ListSelectionListener listener) | Înregistrează un listener de tip ListSelectionListener |
| void clearSelection() | Anulează selecţia curentă |
| int getFirstVisibleIndex() | Obţine indexul primului element vizibil |

| | |
|---|---|
| int getLastVisibleIndex() | Obține indexul ultimului element vizibil |
| int getSelectedIndex() | Obține index-ul elementului selectat |
| Object getItemAt(int index) | Obține elementul din listă de la indexul indicat |
| int getItemCount() | Obține numărul de elemente din listă |
| int[] getSelectedIndices() | Returnează un array cu indecşii elementelor selectate |
| Object getSelectedValue() | Returnează prima valoare selectată |
| Object[] getSelectedValues() | Returnează un tablou cu toate valorile selectate |
| int getVisibleRowCount() | Returnează numărul (preferat) de linii vizibile |
| boolean isSelectedIndex(int index) | Returnează **true** dacă indexul indicat este selectat |
| void setSelectedIndex(int index) | Selectează o singură celulă |
| void setSelectedIndeces(int[ ] indices) | Selectează un set de celule |
| void setSelectedValues(Object anObject, boolean shouldScroll) | Selectează obiectul indicat din listă |
| void setSelectionInterval(int anchor, int lead) | Selectează intervalul indicat |
| ListModel getModel() | Obține modelul de date care păstrează lista de elemente |
| void setListData(Object[ ] listData) | Construieşte un model de listă dintr-un tablou de obiecte şi aplică metoda setModel() |
| Void setListData(Vector listData) | Construieşte un model de listă dintr-un vector şi aplică metoda setModel() |
| void setModel(ListModel model) | Stabileşte un model care va reprezenta conținutul listei |
| void setVisibleRowCount(int visibleRowCount) | Stabileşte numărul (preferabil) de linii care vor fi afişate fără scrollbar |
| void setSelectionMode(int selectionMode) | Determină dacă se poate face doar selecție simplă sau multiplă |
| void setCellRenderer(ListCellRenderer cellRenderer) | Stabileşte obiectul cărui îi va fi delegată sarcina afişării elemetelor |

Putem construi o listă simplă în felul următor:

```
public class ExempluJList extends JFrame{
  JList jlist;
  JLabel status = new JLabel("Stare");
    public ExempluJList() {
      Container contentPane = getContentPane();
      String[] elemente = new String[12];

      for (int i = 0; i< elemente.length; i++){
        elemente[i] = "Elementul "+ i;
      }
      jlist = new JList(elemente);
      JScrollPane jspane = new JScrollPane(jlist);
      jlist.setVisibleRowCount(5);
      jlist.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
```

```
            jlist.addListSelectionListener(new ListSelectionListener(){
              public void valueChanged(ListSelectionEvent e){
                //schimbareValoare(e);
                  selectieElemente(e);
              }
            });
            //contentPane.setLayout(new FlowLayout());
            contentPane.add(jspane, BorderLayout.CENTER);
            JPanel statusBar = new JPanel();
            statusBar.setBorder(BorderFactory.createBevelBorder(
                javax.swing.border.BevelBorder.LOWERED));
            statusBar.setLayout(new FlowLayout());
            statusBar.add(status);
            contentPane.add(statusBar, BorderLayout.SOUTH);
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            setSize(200, 200);
        }
        public void schimbareValoare(ListSelectionEvent e){
        }
        public void selectieElemente(ListSelectionEvent e){
            int [] indecsi = jlist.getSelectedIndices();
            String text_ = "ai selectat ";
            for (int i = 0; i< indecsi.length; i++){
              text_ += " elementul " + indecsi[i];
            }
            //showStatus(text_);
            status.setText(text_);
        }
        public static void main(String[] args) throws Exception{
            UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
            new ExempluJList().show();
        }
    }
```

Stabilirea modului de selcție se face prin metoda *setSelectionMode()* care poate primi următoarele valori:

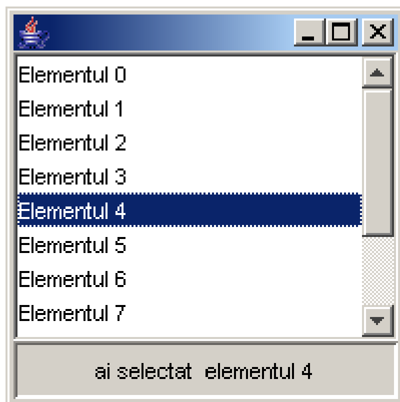
- SINGLE_SELECTION
- SINGLE_INTERVAL_SELECTION
- MULTIPLE_INTERVAL_SELCTION

În acest sens putem adăuga următoarea instrucțiune în codul din listing-ul de mai sus:

```
    jlist.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
```

Determinarea elementelor selectate se poate face folosind metoda *getSelectedIndices()* care returnează un tablou conținând elementele selectate. În acest context putem completa metoda *selectieElemente()* astfel:

```
    public void selectieElemente(ListSelectionEvent e){
      int [] indecsi = jlist.getSelectedIndices();
      String text_ = "ai selectat ";
      for (int i = 0; i< indecsi.length; i++){
        text_ += " elementul " + indecsi[i];
      }
      showStatus(text_);
    }
```




Pentru a extinde capabilităţile unei liste putem modifica modelul de date implicit al acesteia astfel încât să permită spre exemplu şi afişarea unor imagini. Pentru a realiza o astfel de listă va trebui ca mai întâi să mai facem câteva operaţii preliminare, nu chiar foarte simple, dar absolut necesare:

1. În primul rând trebuie să construim un model de date care să stea „în spatele” listei şi care să permită inclusiv folosirea unor imagini. În acest scop construim clasa următoare:

```
// definim mai intai un model de date care sa permita si includerea de imagini
// prin intermediul unr obiecte ImageIcon
// modelExtins trebuie sa implementeze interfata ListModel, dar
// pentru a nu defini toate specificatiile interfetei ne-am folosit
// de clasa DefaultListModel care implementeaza ListModel
class modelExtins extends DefaultListModel{
  public modelExtins(){
    // in modelul de date adaugam imagini folosind metoda addElement
    // mostenita bineninteles din clasa DefaultListModel
    // fiecare element al listei nu va fi altceva decat un vector
    // ce contine doua elemente: un String si un ImageIcon
    this.addElement(new Object[] {"help", new ImageIcon("D:\\ POO_2004\\ProCurs\\img\\jpg\\5.jpg")});
    this.addElement(new Object[] {"find", new ImageIcon("D:\\ ProCurs\\img\\jpg\\3.jpg")});
    this.addElement(new Object[] {"exit", new ImageIcon("D:\\ ProCurs\\img\\jpg\\4.jpg")});
    this.addElement(new Object[] {"save", new ImageIcon("D:\\ ProCurs\\img\\jpg\\7.jpg")});
    this.addElement(new Object[] {"delete", new ImageIcon("D:\\ ProCurs\\img\\jpg\\2.jpg")});
  }
}
```

2. Elementele care formează modelul de date obţinut din clasa *modelExtins* trebuie afişate într-un anume fel (nu putem folosi modelul de afişare implicit pentru că sunt implicate şi obiecte ImageIcon care desemnează imagini). Din acest motiv vom construi un model de prezentare special prin clasa *renderExtins*:

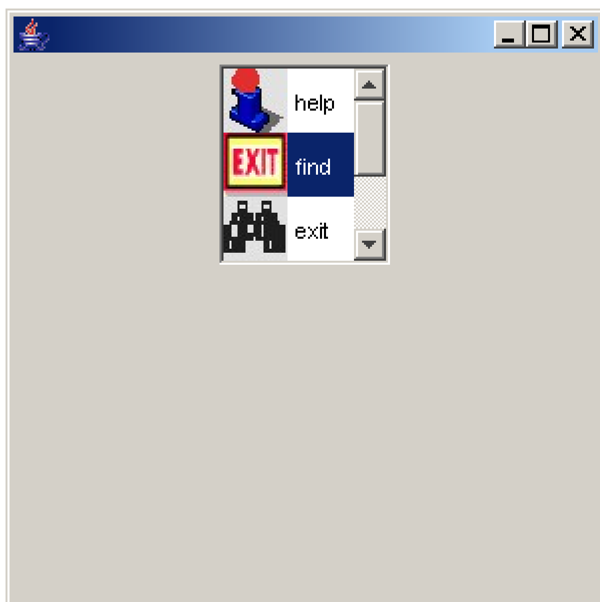
```
// modelul de date obtimut prin clasa modelExtins va fi prezentat
// prin intermediul unui model de afisare/prezentare ce va fi construit
// din clasa renderExtins care deriva din JLabel. Prin urmare elementele
// din modelul de date vor fi afisate de fapt ca etichete (JLabel)
// Pentru a folosi acest model ca render pentru celulele din lista,
// el trebui sa implementeze interfata ListCellRenderer,
// adica trebuie sa contina metoda getListCellRendererComponent
class renderExtins extends JLabel implements ListCellRenderer{
  public renderExtins(){
    this.setOpaque(true);
  }
  public Component getListCellRendererComponent(JList jlist, Object obj,
    int index, boolean isSelected, boolean focus){
// metoda getListCellRendererComponent returneaza
// componenta grafica de afisare - eticheta renderExtins
// si primeste ca argument lista si obiectul din lista care ar trebui afisat
```

```
// modelul de afisare va fi de fapt o eticheta continind un string insotit de o
// imagine - Icon - preluate din modelul de date
      setText((String)((Object[])obj)[0]);
      setIcon((Icon)((Object[])obj)[1]);
// pentru evidentierea afisarii elementelor selectate/deselectate
// se foloseste modul implicit al listei
      if (!isSelected){
        setBackground(jlist.getBackground());
        setForeground(jlist.getForeground());
      }
      else {
        setBackground(jlist.getSelectionBackground());
        setForeground(jlist.getSelectionForeground());
      }
      return this;
  }
}
```

3. Modelul de date (DefaultListModel) şi modelul de afişare (ListCellRender) vor fi valorificate într-o listă a cărei principal merit este că va putea afişa şi imagini:

```
public class ExempluJListExtins extends JFrame{
  JList jlist;
  JLabel status = new JLabel("Stare");
    public ExempluJListExtins() {
        Container contentPane = getContentPane();
        modelExtins model = new modelExtins();
        renderExtins render = new renderExtins();
        jlist = new JList(model);
        jlist.setCellRenderer(render);
        jlist.setVisibleRowCount(3);
        JScrollPane jspane = new JScrollPane(jlist);
        jlist.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
        contentPane.setLayout(new FlowLayout());
        contentPane.add(jspane);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 300);
    }
    public static void main(String[] args) throws Exception{
        UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
        new ExempluJListExtins().show();
    }
}
```

Rezultatul (speram multumitor pentru moment) este următorul:




Unul dintre cele mai populare controale este însă **combo-box**-ul datorită posibilității afişării unui câmp simplu (eventual editabil) însoțit însă de un butom special care determină afişarea unei liste ascunse din care se poate face o selecție. Adică minimum de spațiu cu maximum de efect şi eficiență. Biblioteca Swing aduce pentru implementarea acestui tip de control componenta *JComboBox* a cărei diagramă de moştenire este următoarea:

```
java.lang.Object
| ___ java.awt.Component
        | ___ java.awt.Container
                | ___ javax.swing.JComponent
                        | ___ javax.swing.JComboBox
```

Principalii constructori prin care se pot obține astfel de componente sunt următorii:

| ***Constructor*** | ***Acțiune*** |
|---|---|
| JComboBox() | Construieşte un combo-box simplu |
| JComboBox(ComboBoxModel dataModel) | Construieşte un obiect JComboBox care afişează elementele din lista asociată după modelul de date specificat |
| JComboBox(Object [ ] items) | Construieşte un obiect JComboBox care afişează obiectele în lista asociată din tabloul indicat |
| JComboBox(Vector items) | Construieşte un obiect JComboBox care afişează elementele din vectorul indicat |

Principalele metode prin care un astfel de combo-box poate fi manipulat la nivel minimal (de bază) sunt:

| ***Metoda*** | ***Acțiune*** |
|---|---|
| void addActionListener(ActionListener l) | Înregistrarea unui *action listener* |
| void addItem(Object anObject) | Adăugarea unui element in listă |

| | |
|---|---|
| void insertItemAt(Object anObject, int index) | Inserează un element în listă la poziția (indexul) indicat |
| void addItemListener (ItemListener listener) | Înregistrarea unui listener pentru evenimentele asociate elementelor din listă |
| void configureEditor(ComboBoxEditor anEditor, Object anItem) | Stabileşte un editor pentru elementele din listă (în cazul în care combo-box-ul este editabil) |
| ComBoxEditor getEditor() | Obține editorul folosit pentru a edita elementelul selectat din câmpul obiectului JComboBox |
| int getItemCount() | Obține numărul de elemente din listă |
| int getSelectedIndex() | Obține indexul elementului selectat |
| Object getSelectedItem() | Returnează elementul curent selectat |
| Object[] getSelectedObjects() | Returnează un tablou cu toate elementele selectate |
| void hidePopup() | Închide fereastra ce afişează lista asociată combo-box-ului |
| boolean isEditable() | Returnează true dacă obiectul JComboBox este editabil |
| void removeItem(Object anObject) | Şterge un item din listă |
| void removeAllItem(Object anObject) | Şterge toate elementele din listă |
| protected void selectedItemChanged() | Apelată când este modificat elemetul selectat |
| void setEditable(boolean aFlag) | Stabileşte dacă obiectul JComboBox este editabil |
| void setEditor(ComboBoxEditor anEditor) | Stabileşte un editor pentru elementul selectat în câmpul combo-box-ului |
| void setEnabled(boolean b) | Stabileşte dacă obiectul JComboBox este activat sau dezactivat |
| void setSelectedIndex(int index) | Selectează elementul specificat prin indexul respectiv |
| void setSelectedItem(Object anObject, boolean shouldScroll) | Selectează obiectul indicat din listă |
| void showPopup() | Determină afişarea listei asociate combo-box-ului |
| void setSelectionInterval(int anchor, int lead) | Selectează intervalul indicat |
| ComboBoxModel getModel() | Obține modelul de date care păstrează lista de elemente |
| void setModel(ComboBoxModel model) | Stabileşte un model care va reprezenta conținutul listei elementelor obiectului JComboBox |
| void setMaximumRowCount(int count) | Stabileşte numărul (maxim) de linii care vor fi afişate de către obiectul JComboBox |
| void setRenderer(ListCellRenderer aRenderer) | Stabileşte modelul prin care vor fi afişate elementele |

Cel mai simplu combo-box ar putea fi obținut astfel:

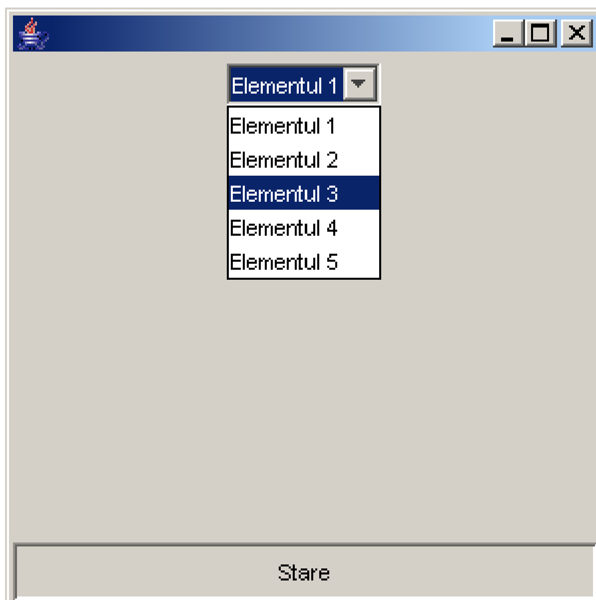
```
public class ExempluJComboBox extends JFrame{
   JComboBox jcombo = new JComboBox();
   JLabel status = new JLabel("Stare");
   public ExempluJComboBox() {
      Container contentPane = getContentPane();
      jcombo.addItem("Elementul 1");
      jcombo.addItem("Elementul 2");
      jcombo.addItem("Elementul 3");
      jcombo.addItem("Elementul 4");
      jcombo.addItem("Elementul 5");
      //contentPane.setLayout(new FlowLayout());
      JPanel mainPane = new JPanel(new FlowLayout());
      mainPane.add(jcombo);
      contentPane.add(mainPane, BorderLayout.CENTER);
      JPanel statusBar = new JPanel();
      statusBar.setBorder(BorderFactory.createBevelBorder(
         javax.swing.border.BevelBorder.LOWERED));
      statusBar.setLayout(new FlowLayout());
      statusBar.add(status);
      contentPane.add(statusBar, BorderLayout.SOUTH);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setSize(300, 300);
   }
   public static void main(String[] args) throws Exception{
      UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
      new ExempluJComboBox().show();
   }
}
```

În mod implict ***evenimentul asociat unei liste*** este ***selectarea unui element***, eveniment care poate fi tratat ca *ActionEvent*. În acest scop putem înregistra un *ActionListener* pentru o listă care să trateze în un eveniment *ActionEvent* într-o metodă *actionPerformed*. În exemplul următor am delegat această responsabilitate către o metodă a applet-ului prin care exemplificăm folosirea unui obiect JComboBox:

```
public class ExempluJComboBox extends JFrame{
   JComboBox jcombo = new JComboBox();
   JLabel status = new JLabel("Stare");
   public ExempluJComboBox() {
      Container contentPane = getContentPane();
      jcombo.addItem("Elementul 1");
      jcombo.addItem("Elementul 2");
      jcombo.addItem("Elementul 3");
      jcombo.addItem("Elementul 4");
      jcombo.addItem("Elementul 5");
      // pentru tratarea evenimentelor
      jcombo.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
          selectareCombo(e);
        }
      });
      //contentPane.setLayout(new FlowLayout());
      JPanel mainPane = new JPanel(new FlowLayout());
      mainPane.add(jcombo);
      contentPane.add(mainPane, BorderLayout.CENTER);
      JPanel statusBar = new JPanel();
      statusBar.setBorder(BorderFactory.createBevelBorder(
         javax.swing.border.BevelBorder.LOWERED));
      statusBar.setLayout(new FlowLayout());
      statusBar.add(status);
      contentPane.add(statusBar, BorderLayout.SOUTH);
```

```
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 300);
    }
    public void selectareCombo(ActionEvent event){
        JComboBox cmb;
        cmb = (JComboBox)event.getSource();
        status.setText("Selectat " + (String)cmb.getSelectedItem());
    }
    public static void main(String[] args) throws Exception{
        UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
        new ExempluJComboBox().show();
    }
}
```

Rezultatul va fi următorul:




Evenimentele pe care le poate produce un combo-box sunt însă mai complexe şi pot face referire distinct la selectarea, respectiv deselectarea, anumitor elemente din lista asociată. Pentru acest lucru trebui mai întâi înregistrat un obiect *ItemListener* care să permită tratarea într-o metodă *itemStateChange()* a evenimentelor *ItemEvent*. Un astfel de eveniment va fi produs atât de către elementelul selectat cât şi de cel care era elementul curent anterior (şi care a fost prin urmare deselectat). Modificăm în acest sens codul anterior astfel:

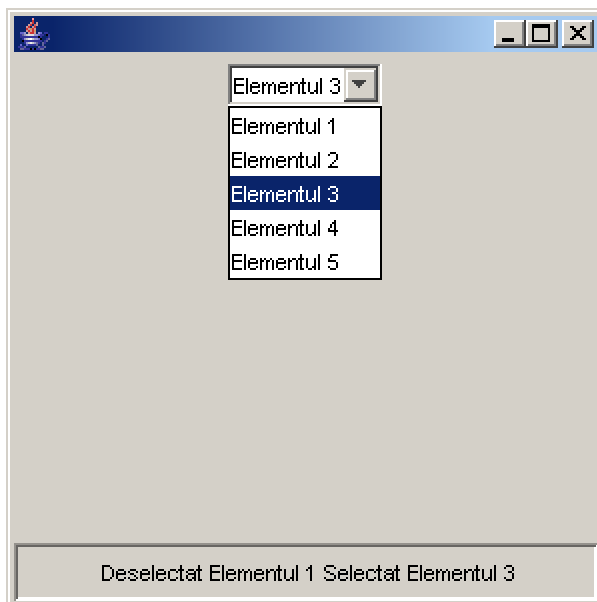
```
public class ExempluJComboBoxExtins extends JFrame{
    JComboBox jcombo = new JComboBox();
    JLabel status = new JLabel("Stare");
    /** Creates a new instance of ExempluJComboBox */
    public ExempluJComboBoxExtins() {
        Container contentPane = getContentPane();

        jcombo.addItem("Elementul 1");
        jcombo.addItem("Elementul 2");
        jcombo.addItem("Elementul 3");
        jcombo.addItem("Elementul 4");
        jcombo.addItem("Elementul 5");
        // pentru tratarea evenimentelor
        jcombo.addItemListener(new ItemListener(){
          public void itemStateChanged(ItemEvent e){
            schimbareElementCombo(e);
          }
        });
```

```
        //contentPane.setLayout(new FlowLayout());

        JPanel mainPane = new JPanel(new FlowLayout());
        mainPane.add(jcombo);
        contentPane.add(mainPane, BorderLayout.CENTER);
        JPanel statusBar = new JPanel();
        statusBar.setBorder(BorderFactory.createBevelBorder(
           javax.swing.border.BevelBorder.LOWERED));
        statusBar.setLayout(new FlowLayout());
        statusBar.add(status);
        contentPane.add(statusBar, BorderLayout.SOUTH);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 300);
    }
    public void schimbareElementCombo(ItemEvent e){
        if (e.getStateChange() == ItemEvent.DESELECTED){
           status.setText(" Deselectat " + (String)e.getItem());
        }
        if (e.getStateChange() == ItemEvent.SELECTED){
           status.setText(status.getText() + " Selectat " + (String)e.getItem());
        }
    }
    public static void main(String[] args) throws Exception{
        UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
        new ExempluJComboBoxExtins().show();
    }
}
```

Iar efectul va fi de aceasta dată:




O altă problemă importantă ar fi ***cum se creează combo-box-urile editabile*** . Pentru a realiza acest lucru avem nevoie de metoda *setEditable()*. De asemenea evenimentele declanşate prin editarea unui anumit element sunt tratate de către un listener-ul înregistrat pentru *editorul* asociat com-box-ului. Acest editor poate fi obţinut prin metoda *getEditor()*, iar asocierea unui listener acestuia se va face deci prin *getEditor().addActionListener()*. În acest sens am modificat exemplul din listingul anterior astfel:

```
public class ExempluJComboBoxEditabil extends JFrame{
    JComboBox jcombo = new JComboBox();
    JLabel status = new JLabel("Stare");
    /** Creates a new instance of ExempluJComboBox */
```

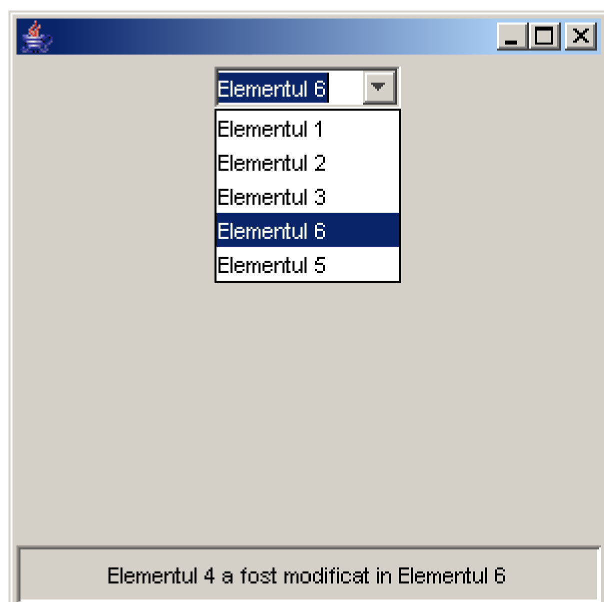
```
    public ExempluJComboBoxEditabil() {
        Container contentPane = getContentPane();
        jcombo.addItem("Elementul 1");
        jcombo.addItem("Elementul 2");
        jcombo.addItem("Elementul 3");
        jcombo.addItem("Elementul 4");
        jcombo.addItem("Elementul 5");
        // pentru declararea editabila a combo-box-ului
        jcombo.setEditable(true);
        // pentru tratarea evenimentelor
        jcombo.getEditor().addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                editareCombo(e);
            }
        });
        //contentPane.setLayout(new FlowLayout());
        JPanel mainPane = new JPanel(new FlowLayout());
        mainPane.add(jcombo);
        contentPane.add(mainPane, BorderLayout.CENTER);
        JPanel statusBar = new JPanel();
        statusBar.setBorder(BorderFactory.createBevelBorder(
            javax.swing.border.BevelBorder.LOWERED));
        statusBar.setLayout(new FlowLayout());
        statusBar.add(status);
        contentPane.add(statusBar, BorderLayout.SOUTH);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 300);
    }
    public void editareCombo(ActionEvent event){
        status.setText((String)jcombo.getSelectedItem() +
          " a fost modificat in " + jcombo.getEditor().getItem());
        Object elementModificat = jcombo.getEditor().getItem();
        Object elementEliminat = jcombo.getSelectedItem();

        jcombo.insertItemAt(elementModificat, jcombo.getSelectedIndex());
        // sau (mai complicat) inlocuim linia de mai sus cu
        // urmatoarele doua linii

        // jcombo.addItem(elementModificat);
        // jcombo.setSelectedItem(elementModificat);
        jcombo.removeItem(elementEliminat);
    }
    public static void main(String[] args) throws Exception{
        UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
        new ExempluJComboBoxEditabil().show();
    }
}
```

În exemplul anterior după editarea unui element acesta va fi îndepărtat şi re-adăugat preluând noua valoare (modificată în editor).

Construirea de combo-box-uri care să prezinte şi imagini prin obiecte gen *ImageIcon* se face în mod similar cu procedeul folosit în cazul listelor (obiectelor de tip *JList*) prezentate mai înainte.




**Controale pentru derularea suprafețelor neafişate în întregime**

Spațiul a constituit întotdeauna o limită esențială în construirea interfețelor grafice. Din acest motiv au apărut o serie de controale al căror principal rol este să prezinte dintr-o suprafață de afişare doar anumite fragmente sau porțiuni pe de o parte, şi pe de altă parte să permită derularea pentru a face vizibile alte porțiuni sau fragmente din accealaşi „document".

În Java principalele componente responsabile pentru asemenea sarcini sunt (din Swing): *JViewport*, *JScrollPane, JScrollBar*.

```
java.lang.Object
| ___ java.awt.Component
        | ___ java.awt.Container
                | ___ javax.swing.JComponent
                        | ___ javax.swing.JViewPort
                        | ___ javax.swing.JScrollPane
                        | ___ javax.swing.JScrollBar
```

Probabil cea mai des folosită componentă este *JScrollPane*, motiv pentru care o vom prezenta în continare.

Un *JScrollPane* poate fi construit „gol", urmând ca ulterior să-i fie adăugate componente, sau conținând de la început o componentă, funcție de constructorii folosiți:

| ***Constructor*** | ***Acțiune*** |
|---|---|
| JScrollPane() | Crează un scroll pane fără conținut (gol). Conținutul poate fi specificat ulterior – fiind un Container deci va moşteni metodele *add()* |
| JScrollPane(Component view) | Creează un obiect *JScrollPane* care afişează conținutul componentei specificate |
| JScrollPane(Component view, int vsbPolicy, int hsbPolicy) | Construieşte un obiect *JScrollPane* care afişează conținutul componentei specificate şi a cărei porțiune |

| | |
|---|---|
| | vizibilă poate fi controlată printr-o pereche de bare de derulare (scroll bars) |
| JScrollPane(vsbPolicy, int hsbPolicy) | Crează un scroll pane fără conținut (gol) având specificate modul de utilizare al barelor de derulare (scrollbar policies) |

Principalele respondabilități se referă deci la afişarea tipului de bare de defilare, la politica privind afişarea acestor (niciodată, doar când este necesar, întotdeauna) astfel că o parte dintre metodele necesare în acest sens sunt următoarele:

| ***Metoda*** | ***Acțiune*** |
|---|---|
| JScrollBar createHorizontalScrollBar() | Creează o bară de derulare orizontală |
| JScrollBar getHorizontalScrollBar() | Obține bara de derulare orizontală |
| int getHorizontalScrollBarPolicy() | Obține o valoare prin care este indicată politica privind afişarea barelor de derulare orizontale |
| JScrollBar createVerticalScrollBar() | Creează o bară de derulare verticală |
| JScrollBar getVerticalScrollBar() | Obține bara de derulare vericală |
| int getVerticalScrollBarPolicy() | Obține o valoare prin care este indicată politica privind afişarea barelor de derulare verticale |
| void setHorizontalScrollBar(JScrollbar horizontalScrollBar) | Stabileşte bara de derulare orizontală |
| void setHorizontalScrollBarPolicy(int Policy) | Stabileşte politica privind afişarea barelor de derulare orizontale |
| void setVerticalScrollBar(Jscrollbar verticalScrollBar) | Stabileşte bara de derulare verticală |
| void setVerticalScrollBarPolicy(int Policy) | Stabileşte politica privind afişarea barelor de derulare verticale |

Ca exemplu vom construi un panou organizat ca un grid care are prea multe linii şi prea multe coloane pentru a putea fi vizualizat în întregime. Pentru a rezolva criza de spațiu recurgem la un *JScrollPane* cu bare de derulare (sau defilare) verticale şi orizontale:

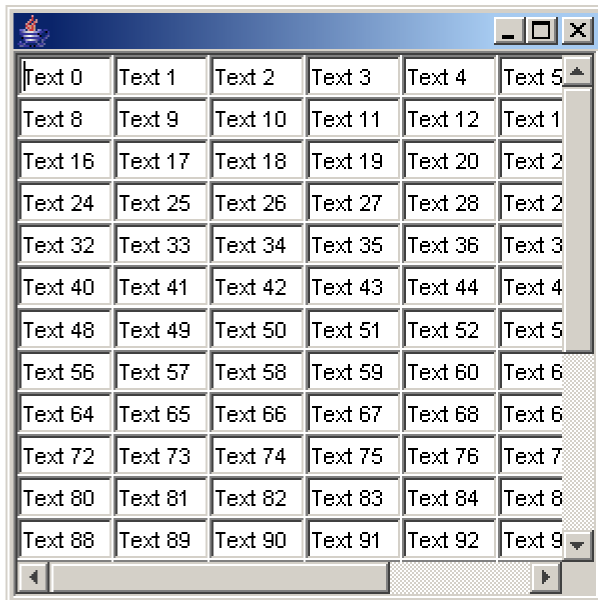
```
public class ExempluJScrollPane extends JFrame{
   public ExempluJScrollPane() {
      Container contentPane = getContentPane();
      JPanel jpanel = new JPanel();
      jpanel.setLayout(new GridLayout(20, 16));
      for (int i = 0; i<=150; i++)
        jpanel.add(new JTextField("Text " + i));

      JScrollPane jspane = new JScrollPane(jpanel,
         ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
         ScrollPaneConstants.HORIZONTAL_SCROLLBAR_ALWAYS);

      contentPane.add(jspane);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setSize(300, 300);
   }
   public static void main(String[] args) throws Exception{
```

```
        UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
        new ExempluJScrollPane().show();
    }
}
```

La execuţie vom obţine:



Valorile *int* prin care sunt desemnate politicile în ceea ce priveşte afişarea barelor de defilare se obţin de membrii (constantele) clasei *ScrollPaneConstants* (sau direct din *JScrollPane*):

- **HORIZONTAL_SCROLL_BAR_ALWAYS**
- **HORIZONTAL_SCROLL_BAR_AS_NEEDED**
- **HORIZONTAL_SCROLL_BAR_NEVER**
- **VERTICAL_SCROLL_BAR_ALWAYS**
- **VERTICAL_SCROLL_BAR_AS_NEEDED**
- **VERTICAL_SCROLL_BAR_NEVER**

Obiectele bare de derulare (sau defilare) deşi însoţesc de regulă un scroll pane, pot fi folosite şi de sine stătător şi asociate în mod independent altor componente care ar avea nevoie. Clasa din Swing folosită pentru obţinerea unor astfel de obiecte este *JScrollBar*. Constructorul implicit fără argumente **JScrollBar()** crează un scroll bar orizontal, însă există şi posibilitatea specificării modului de orientare **JScrollBar(int orientation)** sau mai detaliat chiar limitele intervalului de derulare şi pasul **JScrollBar(int orientation, int value, int extent, int min, int max)**.

De asemenea există şi metode pentru obţinerea şi stabilirea:

- orientării: **int getOrientation()** şi **void setOrientation(int orientation)**;
- valorii, limitei minime şi maxime: **int getValue(), int getMinimum()**, **int getMaximum()** şi **void setValue(int value), void setMinimum(int minimum), void setMaximum(int maximum);**

- dimensiunilor (extent – dimensiunea zonei vizibile, mimim, maxim): **int getVisibleAmount(), Dimension getMinimumSize(), Dimension getMaximumSize()** şi **void setVisibleAmount()**;
- modului selectat/deselectat **void setEnabled(boolean x)**.

După cum se poate trage concluzia modelul de date al unui obiect de tip *JScrollBar* are patru proprietăți: *minimum, maximum, valoare, extent* ([extent + valoare]<maximum). Pentru manevrarea obiectelor de tip *JScrollBar* direct prin intermediul modelului de date se pot invoca metodele:

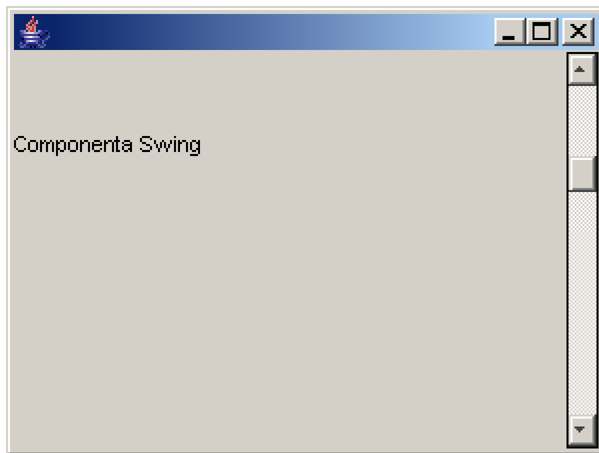
- **BoundedRangeModel getModel()** pentru a obține modelul de date folosit;
- **void setModel(BoundedRangeModel newModel)** pentru a specifica un model de date care să stabilească proprietățile fundamentale;
- **void setValues(int newValue, int newExtent, int newMin, int newMax)** pentru a stabili în mod direct proprietățile modelului de date folosit.

Evenimentele care le poate genera un obiect *JScrollBar* se numesc *AdjustmentEvent* şi pot fi tratate de obiecte ce implementează interfața *AdjustmentListener* care specifică metoda *adjustmentValueChanged()*. Iată un exemplu în acest sens:

```
public class ExempluJScrollBar extends JFrame{
  JScrollBar sbar = new JScrollBar(JScrollBar.VERTICAL, 0, 20, 0, 180);
  JPanelScroll jpanel = new JPanelScroll();
    public ExempluJScrollBar() {
      Container contentPane = getContentPane();
      sbar.setBorder(BorderFactory.createLineBorder(Color.black));
      contentPane.add(jpanel, BorderLayout.WEST);
      contentPane.add(sbar, BorderLayout.EAST);

      sbar.addAdjustmentListener(new AdjustmentListener(){
        public void adjustmentValueChanged(AdjustmentEvent e){
          JScrollBar sb = (JScrollBar)e.getSource();
          jpanel.setScrolledPosition(e.getValue());
          jpanel.repaint();
        }
      });
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setSize(300, 225);
    }
    public static void main(String[] args) throws Exception{
      UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
      new ExempluJScrollBar().show();
    }
}
class JPanelScroll extends JPanel{
    JLabel jlabel = new JLabel("Componenta Swing");
    int y= 0;
    JPanelScroll(){
      add(jlabel);
    }
    public void paintComponent(Graphics g){
      super.paintComponent(g);
      jlabel.setLocation(0, y);
    }
    public void setScrolledPosition(int p){
      y = p;
    }
}
```

Efectul:




#### *2.4.3.3. Definitivarea aplicațiilor*

Aplicațiile cu formulare Java pot îmbrăca forma applet-urilor executabile în mediile runtime Java (JRE) ale browserelor Web, sau pot rula de sine stătătoare dacă sunt derivate din clasa JFrame.

**Crearea unei aplicații simple folosind JFrame. Manipularea Look And Feel-ului**

După cum e uşor de intuit, aplicațiile în JAVA au la bază clasa *JFrame* ale cărei obiecte, spre deosebire de applet-uri, sunt afişate în ferestre proprii. De altfel clasa *JFrame* este derivată din clasa *Window*, obiectele acesteia având coresponenți ferestre (peer) ale platformei de operare pe care rulează (de aceea se mai numesc şi componente *heavyweight* – „grele”):

```
java.lang.Object
| ___ java.awt.Component
      | ___ java.awt.Container
            | ___ java.awt.Window
                  | ___ java.awt.Frame
                        | ___ javax.swing.JFrame
```

Lansarea în execuție a unui *JFrame* în mod direct se poate realiza specificând numele acestuia mediului run-time (JRE), condiția fiind ca să existe o metodă ***public static*** *void main(String [] args)*.

Iată spre exemplu cea mai simplă aplicație cu un *JFrame*:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class AplicatieJFrame {
  public AplicatieJFrame(){
    JFrameSimplu jframe = new JFrameSimplu("Aplicatie");
    jframe.setSize(400, 300);
    jframe.validate();
    jframe.setVisible(true);
```

```
  }
  public static void main(String[] args) {
    AplicatieJFrame app = new AplicatieJFrame();
  }
}

class JFrameSimplu extends JFrame {
  JFrameSimplu(String titlu){
    super(titlu);
    Container contentPane = getContentPane();
    FramePanel framePanel = new FramePanel();
    contentPane.add(framePanel);
  }
}
class FramePanel extends JPanel{
  JLabel jlabel = new JLabel(" Componenta din panel ");
  FramePanel(){
    setBackground(Color.blue);
    setLayout(new FlowLayout());
    jlabel.setForeground(Color.white);
    jlabel.setBorder(BorderFactory.createEtchedBorder());
    add(jlabel);
  }
}
```

Rezultatul ar trebui să fie:




La fel ca şi în cazul Frame-urilor din AWT închiderea aplicaţiei odată cu închiderea ferestrei *JFrame*-ului rămâne nerezolvată în mod implicit. Totuşi în această direcţie biblioteca Swing vine cu o noutate: clasa *JFrame* deţine o metodă specifică ce va fi apelată la închiderea ferestrei, metoda se numeşte ***setDefaultCloseOperation()*** şi poate primi ca argument următoarele constante:

- **DO_NOTHING_ON_CLOSE**
- **HIDE_ON_CLOSE**
- **DISPOSE_ON_CLOSE**
- **EXIT_ON_CLOSE**

Prin urmare închiderea applicației la închiderea ferestrei poate fi realizată adăugând următoarea secvență de cod (scrisă în constructorul *JFrameSimplu*):

```
setDefaultCloseOperation(DISPOSE_ON_CLOSE);
addWindowListener(new WindowAdapter(){
  public void windowClosed(WindowEvent e){
    System.exit(0);
  }
});
```

Sau cel mai simplu, secvența instrucțiuni de mai sus se înlocuişte cu:

```
setDefaultCloseOperation(EXIT_ON_CLOSE);
```

O altă caracteristică deosebită venită odată cu biblioteca Swing este posibilitatea modificării dinamice a *look-and-feel*-ului interfețelor grafice, aşa încât ferestrele să aibă aspectul interfeței Windows, Motif (de la UNIX) sau Metal (aspectul grafic caracteristic JavaSwing). Rolul esențial în această privință îl joacă clasa *UIManager* care deține metoda ***setLookAndFeel()*** ce poate primi următoarele argumente:

- ”javax.swing.plaf.metal.MetalLookAndFeel”
- ”com.sun.java.swing.plaf.motif.MotifLookAndFeel”
- ”com.sun.java.swing.plaf.windows.WindowsLookAndFeel”

Iată codul un aplicații care vizează acest lucru:

```
public class AplicatieSchimbLookAndFeel {
  public AplicatieSchimbLookAndFeel(){
    JFrameApp jframe = new JFrameApp("Aplicatie");
    jframe.pack();
    jframe.validate();
    jframe.setVisible(true);
  }
  public static void main(String[] args) {
    AplicatieSchimbLookAndFeel app = new AplicatieSchimbLookAndFeel();
  }
}

class JFrameApp extends JFrame {
  JFrameApp(String titlu){
    super(titlu);
    Container contentPane = getContentPane();
    PanelJFrameApp framePanel = new PanelJFrameApp();
    contentPane.add(framePanel);
    // pentru inchiderea aplicatiei
    setDefaultCloseOperation(EXIT_ON_CLOSE);
  }
}
class PanelJFrameApp extends JPanel implements ActionListener{
  JRadioButton rb1 = new JRadioButton("Metal");
  JRadioButton rb2 = new JRadioButton("Motif");
  JRadioButton rb3 = new JRadioButton("Windows");
  JRadioButton rb4 = new JRadioButton("Skin");
  JRadioButton rb5 = new JRadioButton("Plastic");
  JRadioButton rb6 = new JRadioButton("PlasticXP");

  PanelJFrameApp(){
    setLayout(new FlowLayout());
    add(new JButton("Buton"));
    add(new JTextField("TextField"));
```

```
    add(new JCheckBox("CheckBox"));
    add(new JRadioButton("Buton Radio"));
    JList jList = new JList(new String []
      {"Element 1", "Element 2", "Element 3", "Element 4"});
    jList.setBorder(BorderFactory.createBevelBorder(javax.swing.border.BevelBorder.LOWERED));
    add(new JScrollPane(jList,
        JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
        JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS));
    JScrollBar jScrollBar = new JScrollBar(JScrollBar.HORIZONTAL, 10, 10, 0, 180);

    add(jScrollBar);

    ButtonGroup grup = new ButtonGroup();
    grup.add(rb1);
    grup.add(rb2);
    grup.add(rb3);
    grup.add(rb4);
    grup.add(rb5);
    grup.add(rb6);

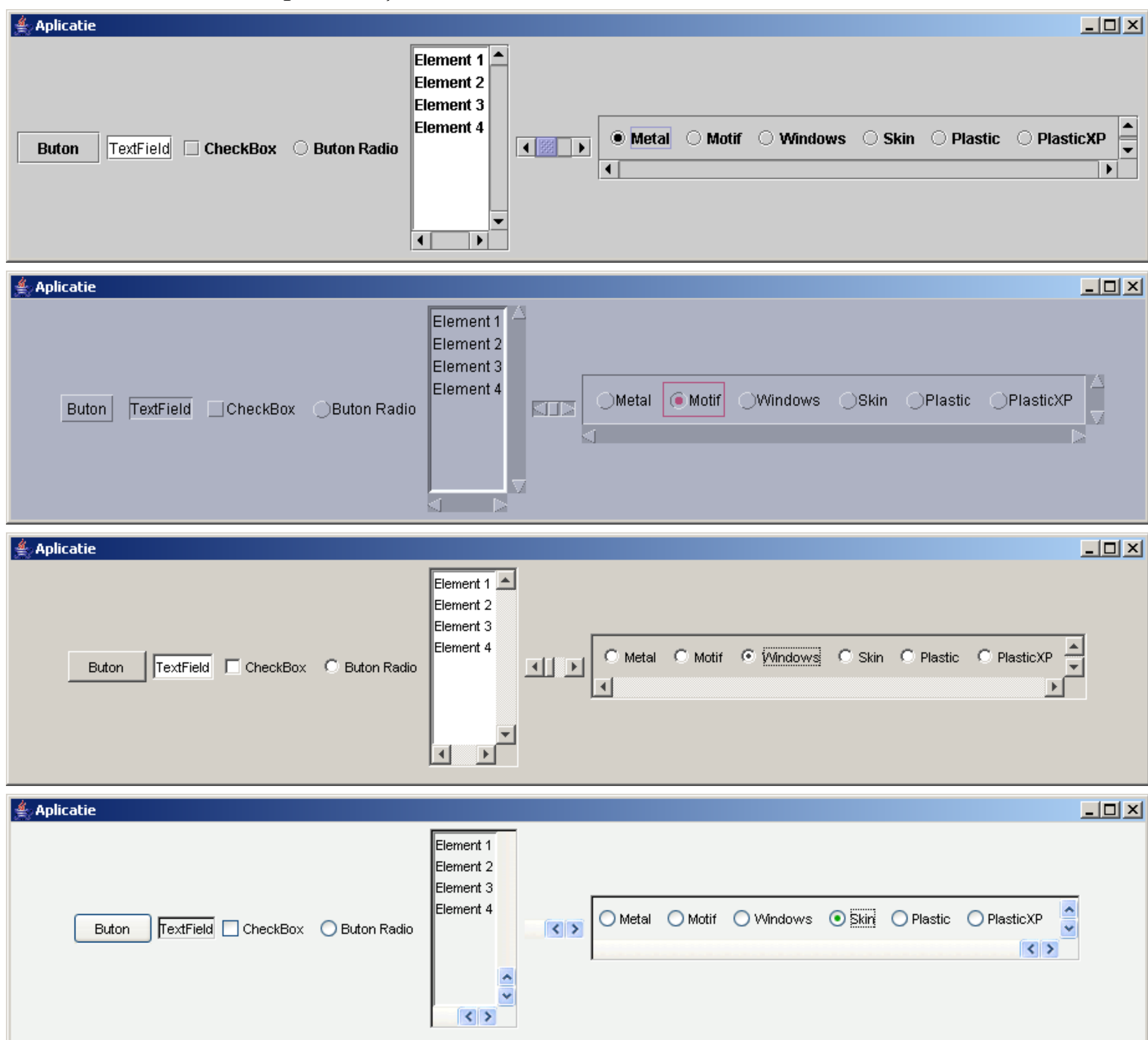
    rb1.addActionListener(this);
    rb2.addActionListener(this);
    rb3.addActionListener(this);
    rb4.addActionListener(this);
    rb5.addActionListener(this);
    rb6.addActionListener(this);

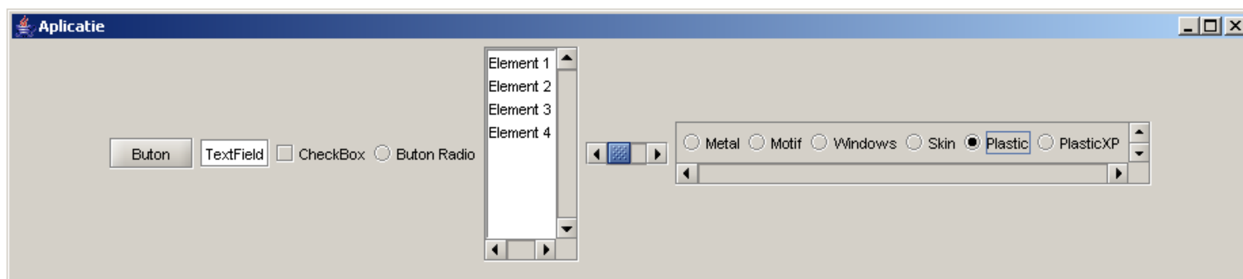
    JPanel panouButoane = new JPanel(new FlowLayout());
    panouButoane.add(rb1);
    panouButoane.add(rb2);
    panouButoane.add(rb3);
    panouButoane.add(rb4);
    panouButoane.add(rb5);
    panouButoane.add(rb6);
    JScrollPane scrollPane = new JScrollPane(panouButoane,
        JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
        JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
    add(scrollPane);

  }
  public void actionPerformed(ActionEvent e){
   try {
      if((JRadioButton)e.getSource() == rb1)
       UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
      if((JRadioButton)e.getSource() == rb2)
        UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
      if((JRadioButton)e.getSource() == rb3)
        UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
      if((JRadioButton)e.getSource() == rb4)
        UIManager.setLookAndFeel("com.l2fprod.gui.plaf.skin.SkinLookAndFeel");
      if((JRadioButton)e.getSource() == rb5)
        UIManager.setLookAndFeel("com.jgoodies.plaf.plastic.Plastic3DLookAndFeel");
      if((JRadioButton)e.getSource() == rb6)
        UIManager.setLookAndFeel("com.jgoodies.plaf.plastic.PlasticXPLookAndFeel");
    }
    catch(Exception excpt){}
    SwingUtilities.updateComponentTreeUI(this);
  }
}
```

După cum se observă codul esenţial în această privinţă rezidă în metoda *actionPerformed()* a JPanel-ului care implementează şi interfaţa care să-i permită tratarea evenimentelor celor trei butoane radio. Aceste metode iniţiază acţiunea de schimbare a *look&feel*-ului instanţei *JFrame*. La o privire mai în detaliu observăm că mai există o instrucţiune esenţială în listingul de mai sus, instrucţiune prin care se invocă metoda *SwingUtilities.updateComponentTree()*. Motivul este următorul: pentru ca schimbările să fie cu adevărat vizibile trebui reactualizate toate componentele afişate. Pentru a nu parcurge individual fiecare componentă am recurs la respectiva metodă care primeşte ca argument o componentă ce poate fi un container ce conţine alte componente ş.am.d. şi poate să le reactualizate pe toate acestea. În cazul nostru toate componentele se găsesc pe *PanelJFrameApp* aşa încât am invocat metoda *SwingUtilities.updateComponentTree()* trimiţându-i ca argument referinţa curentă a acestuia. Dacă am fi plasat componentele direct în JFrame-ul *JFrameApp* metoda ar fi trebuit să primească ca argument content pane-ul acestuia (care poate fi obţinut prin *getContentPane()*).

Ca urmare vom putea obţine următoarele stări:






**Construirea unei fereastra principale a aplicațiilor tip desktop cu meniuri *Swing***

De regulă, ferestrele principale ale aplicațiilor *desktop* reprezintă calea esențială pentru accesul la funcțiile principale ale aplicației. De regulă acest lucru realizat printr-un meniu însoțit, eventual, şi de o bară de instrumente.

***Crearea meniurilor***

Modelul orientat obiect al meniurilor în Swing are următoarele caracteristici:

- bară principală reprezintă o instanță a clasei *JMenuBar*, sau, pentru meniurile contextuale, *JPopUpMenu*;
- meniurile accesebile din bara principală sunt obținute din clasa *JMenu*;
- elementele individuale ale unui menu (instanță *JMenu*) sunt construite pe baza clasei *JMenuItem* care are următoarele subclase: *JCheckBoxMenuItem, JRadioButtonMenuItem* şi *JMenu*. Prin urmare un element individual dintr-un meniu poate fi, la rândul său, un al submenu (instanță *JMenu*);

Ca urmare definiția celui mai simplu meniu ar putea fi următoarea:

```
public class MainMenuBar extends JMenuBar{
    JFrame frmParent;
    public MainMenuBar() {
        JMenu fileMenu = new JMenu("File");
        JMenu editMenu = new JMenu("Edit");
        JMenu quitMenu = new JMenu("Quit");

        // Meniul File: Open, Save, Close
        JMenuItem openItem = new JMenuItem("Open");
        JMenuItem saveItem = new JMenuItem("Save");
        JMenuItem closeItem = new JMenuItem("Close");

        //Meniul Edit: Cut, Copy, Paste, Find (Find, Replace)
        JMenuItem cutItem = new JMenuItem("Cut");
        JMenuItem copyItem = new JMenuItem("Copy");
        JMenuItem pasteItem = new JMenuItem("Paste");
        JSeparator separator1 = new JSeparator();
        JMenu findMenu = new JMenu("Find");
        JMenuItem findItem = new JMenuItem("Find");
        JMenuItem replaceItem = new JMenuItem("Replace");

        //Meniul Quit
        JMenuItem exitItem = new JMenuItem("Exit");
        JMenuItem aboutItem = new JMenuItem("About");

        fileMenu.add(openItem);
        fileMenu.add(saveItem);
        fileMenu.add(closeItem);

        editMenu.add(cutItem);
```

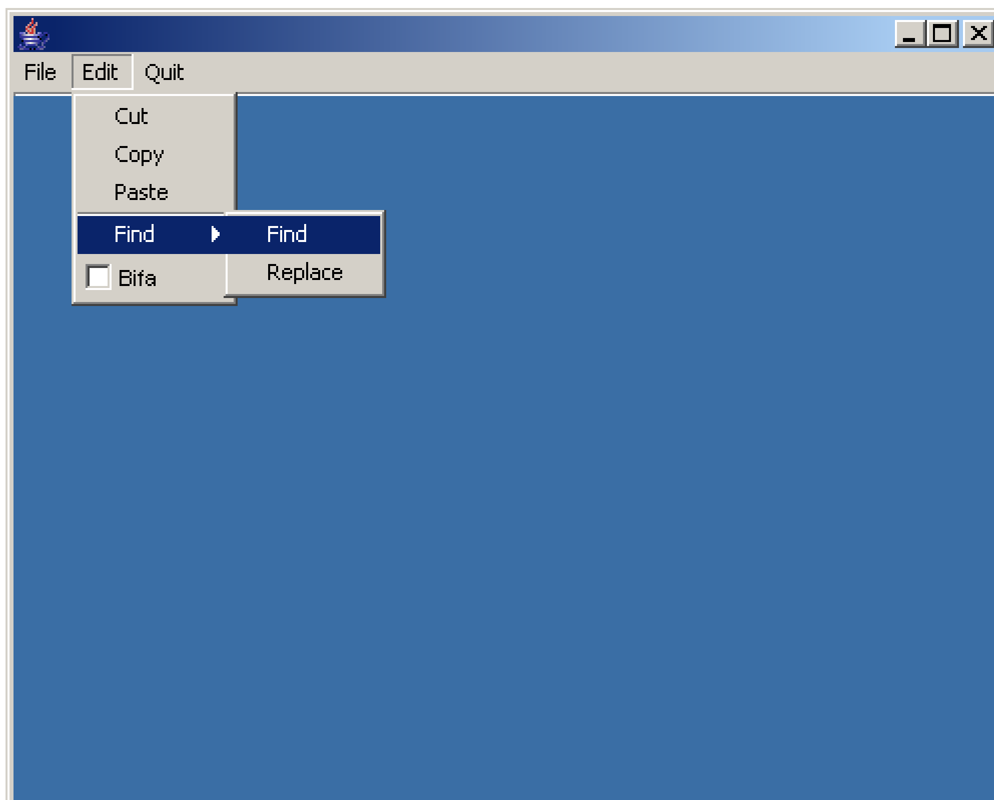
```
            editMenu.add(copyItem);
            editMenu.add(pasteItem);
            editMenu.add(separator1);
            findMenu.add(findItem);
            findMenu.add(replaceItem);
            editMenu.add(findMenu);
            editMenu.add(new JCheckBox("Bifa"));
            quitMenu.add(exitItem);
            quitMenu.add(aboutItem);

            add(fileMenu);
            add(editMenu);
            add(quitMenu);
        }
    }
```

Pentru a dispune un meniu într-un formular, avem nevoie de o instanță *JFrame* pentru bara de meniu, stabilită prin metoda *setJMenuBar()*. De exemplu clasa:

```
    public class FormMain extends JFrame{
        public FormMain() {
            this.setJMenuBar(new MainMenuBar());
            this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            this.setSize(500, 500);
            this.setVisible(true);
        }
        public static void main(String[] args) {
            new FormMain();
        }
    }
```

va avea la runtime următorul rezultat:



***Selecția elementelor din meniuri***

Fiecare element dintr-un meniu (JMenu) răspunde la evenimentele de selecție asemănător butoanelor de comandă (JButton) deoarece clasa *JMenu* este derivată din clasa abstractă *AbstractButton*.

Prin urmare calea cea mai simplă pentru tratarea evenimentelor unui element dintr-un meniu este înregistrarea pentru acea opțiune a unui *ActionListener* care va prelua în metoda *actionPerformed()* evenimente de tip *ActionEvent*. Metoda *getActionCommand()* a unei instanțe *ActionEvent* va returna numele opțiunii care a generat respectivul eveniment. De asemenea, metoda *getSource()* a respectivului eveniment va returna o referință către elementul din meniu care l-a generat. Spre exemplu, dacă dorim ca la selectarea unei opțiuni să afişăm într-o căsuță de dialog numele acesteia şi apoi s-o dezactivăm atunci va trebui să folosim următoarea secvență de cod:

```
JMenuItem findItem = new JMenuItem("Find");
findItem.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent e){
      JOptionPane.showMessageDialog(null, "A fost selectata optiunea \"" +
         e.getActionCommand() + "\"");
      JMenuItem item = (JMenuItem)e.getSource();
      item.setEnabled(false);
   }
});
```

Alte evenimente legate de elementele meniurilor şi care se pot dovedi utile în anumite situații sunt *MenuDragMouseEvent* (generat la trecerea cursorului mouse-ului) şi *MenuKeyEvent*.

O altă caracterisitică asociată optiunilor din meniuri este posibilitatea accesării lor folosind tastatura prin aşa numitele *shortcut*-uri. În această privință există două cazuri:

- acceleratori – combinații de taste care indiferent dacă meniul care conține opțiunea vizată este activat sau nu, va genera selecția acesteia;
- *mnemonice* – o tastă asociată unei o unei opțiuni de meniu care va genera selecția acesteia numai dacă meniul din care face parte este activat.

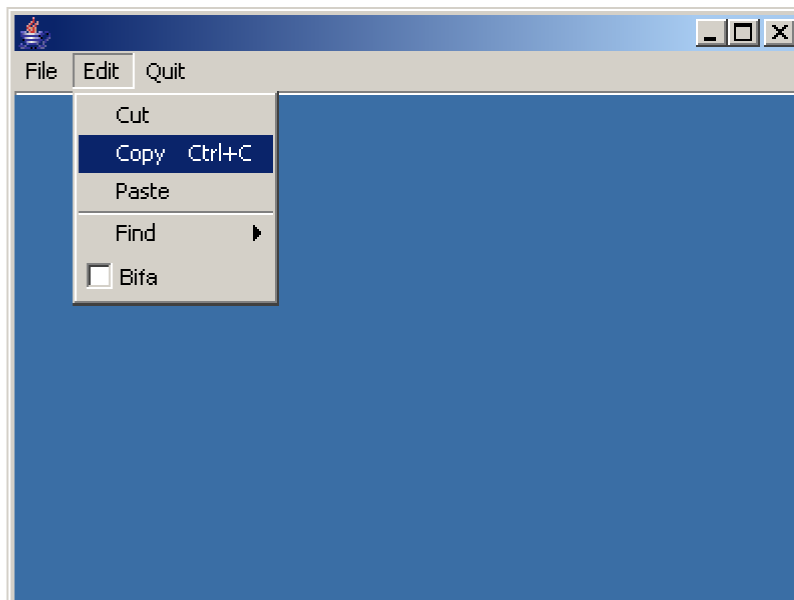
Asocierea unui *accelerator* pentru o opțiune de meniu este o sarcină ceva mai anevoioasă datorită diferitelor paltforme grafice (ale sistemelor de operare) pe care operează Swing. Metoda care joacă rolul esențial în acestă privință este *setAccelerator()* care însă cere drept argument o combinație de taste reprezentată folosind clasa *KeyStroke* care gestionează aspectele particulare ale fiecărei platforme grafice în parte. De exemplu asocierea combinației "CTRL+C" opțiunii *Copy* din meniul de mai sus se face astfel:

```
JMenuItem copyItem = new JMenuItem("Copy");
copyItem.setAccelerator(KeyStroke.getKeyStroke('C',
   Toolkit.getDefaultToolkit().getMenuShortcutKeyMask(), false));
copyItem.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent e){
      JOptionPane.showMessageDialog(null, "A fost selectata optiunea \"" +
         e.getActionCommand() + "\"");
   }
});
```

Acocierea unui *mnemonic* pentru o opțiune de meniu se realizează mult mai simplu, caracterul reprezentând tasta respectivă fiind specificat direct în metoda *setMnemonic()*:

```
copyItem.setMnemonic('C');
```

Efectul ultimelor două acţiuni este următorul:




***Lansarea formularelor modale şi nemodale***

Am arătat, în paragrafele anterioare, că pentru a obţine formulare este necesară clasa *JFrame*. Mai există însă în *Swing* o clasă care poate fi utilizată cu succes pentru obţinerea formularelor: clasa *JDialog*. La fel ca şi *JFrame*, *JDialog* gestionează aspectele legate de obţinerea unei "ferestre" a platformei grafice (a sistemului de operare) unde este instalată aplicaţia. Deosebirea rezidă în faptul că, în vreme ce formularele prezentate cu *JFrame* sunt nemodale, cele prezentate cu *JDialog* pot deveni modale, adică preiau controlul aplicaţiei, iar celelalte formulate (deschise şi rămase "în spate") nu pot fi accesate sau activate decât după închiderea ferestrei *JDialog* active.

Caliatatea modală a unui formular obţinut cu *JDialog* este stabilită fie din constructor:

```
JDialog(Dialog owner, boolean modal);
JDialog(Dialog owner, String title, boolean modal);
JDialog(Frame owner, boolean modal);
JDialog(Frame owner, String title, boolean modal);
```

fie cu ajutorul metodei

```
set Modal(Boolean modal);
```

De exemplu am putea construi un formular care să fie construit modal sau nemodal în felul următor:

```
public class FormularAplicatie extends JPanel{
   public final static int MODAL = 1;
   public final static int MODELESS = 2;

   private Window form;
   private int windowType = MODAL;
   /** Creates a new instance of FormularAplicatie */
   public FormularAplicatie() {
   }
   public void setWindowType(int type){
      if (type == 1 || type == 2)
         windowType = type;
```

```
        }
        public void activate(){
            initComponents();
            if (windowType == MODELESS){
                form = new JFrame();
                ((JFrame)form).setContentPane(this);
                ((JFrame)form).setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
            }else {
                form = new JDialog();
                ((JDialog)form).setModal(true);
                ((JDialog)form).setContentPane(this);
                ((JDialog)form).setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
            }
            form.setSize(250, 250);
            form.setVisible(true);

        }
        private void initComponents(){
            this.setLayout(new FlowLayout());
            button = new JButton("Inchide formularul");
            button.addActionListener(new ActionListener(){
                public void actionPerformed(ActionEvent e){
                    form.dispose();
                }
            });
            label = new JLabel("Acesta este un formular " +
                (windowType==MODAL?"modal":"nemodal"));
            add(label);
            add(button);
        }
        private JButton button;
        private JLabel label;

    }
```

După cum se observă din listingul de mai sus, formularul nostru este construit pe baza unui *JPanel* căruia i-am creat un membru specific *windowType* care relevă calitatea modală sau nemodală a acestuia. Modificarea acestei proprietăți se poate realiza folosind metoda *setWindowType*. În metoda *activate()* se observă că instanțierea clasei suport pentru *JPanel,* ce conține componentele formularului, se face funcție de valoarea proprietății *windowType*. Astfel, dacă *windowType* indică un formular nemodal se foloseşte un *JFrame* obişnuit, în caz contrar se foloseşte un *JDialog*.

Pentru invocarea acestui formular am folosit opțiunile *Open* şi *Save* din meniul anterior, astfel că am adăugat în constructorul acestuia şi următoarea secvență de cod:

```
            openItem.addActionListener(new ActionListener(){
                public void actionPerformed(ActionEvent e){
                    FormularAplicatie frm = new FormularAplicatie();
                    frm.setWindowType(FormularAplicatie.MODAL);
                    frm.activate();
                }
            });
            saveItem.addActionListener(new ActionListener(){
                public void actionPerformed(ActionEvent e){
                    FormularAplicatie frm = new FormularAplicatie();
                    frm.setWindowType(FormularAplicatie.MODELESS);
                    frm.activate();
                }
            });
```

La execuție ar trebui să obținem următorul rezultat: dacă vom invoca opțiunea *Open* vom obține formularul modal şi nu vom mai putea accesa formularul care deține meniul principal:

